MATH 540 FALL 2021

GIDEON SIMPSON

Contents

Date: August 2021.

1. Overview

The purpose of Math 540 is to introduce concepts from scientific computing. Scientific computing sits at the interface of:

Numerical Analysis: Numerical analysis is the mathematically rigorous study of algorithms. This includes topics like the stability and convergence of numerical methods.

Software Engineering: Software engineering is a skill set for writing well written, well organized, code. Ideally, such codes have a modular structure allowing for easy reuse and extension in other problems.

Applications: Motivating the use of numerical methods is some underlying application, requiring the approximation of some quantity of interest, such as the solution of a partial differential equation (PDE).

Scientific computing can be viewed as the synthesis of these three topics.

Codes for this course are distributed from `https://github.com/gideonsimpson/Math_540_2021`.

2. Nuts & Bolts

2.1. Languages.

Matlab: For many years, Matlab has been **the** go to software environment/language for rapid scientific computing. It is well established and documented, with a large number of both official and unofficial supplemental codes. For problems that are "small enough" to fit on a desktop computer, it may, justifiably, be a good place to start. In particular, for small scale linear algebra problems (solving systems, performing matrix decompositions, *etc.*), other languages are unlikely to outperform Matlab. It is a "low overhead" language, in that the number of lines of code one must write to solve and interpret a particular problem is modest, particularly in comparison to a language like C.

However, Matlab has several weaknesses. First, it is commercial software, requiring a license. This can make it difficult to share code with collaborators.

Second, at least historically, codes which required more sophisticated data structures (things that might be sensibly be defined as classes in C++ or Java) tended to offer poor performance. There were also performance issues with advanced control structures (*i.e.* loops containing if-then-else statements), though this has been improved.

Python: Python is a widely available free open source language that, like Matlab, has a comparatively low barrier to entry. When combined with NumPy, SciPy, and matplotlib, Python can accomplish many of the same tasks as Matlab with comparable performance, and without significantly more complicated coding.

With a modest amount of work, Fortran and C/C++ libraries can be linked to Python and used inside of Python codes. A prominent example is FEniCS, `https://fenicsproject.org/`, a finite element library that has a Python interface sitting on top of the underlying C++ code. Other important libraries that can be used from within Python include MPI and PETSc.

Python is also a broadly used general purpose language, and it inherents modern programming paradigms like classes and modules.

A weakness of Python is that optimal performance may require the use of other auxiliary packages, like numba and NumExpr. At times, it will be necessary to rewrite particular pieces of the code in C/C++ and then bind those portions to Python.

Julia: Julia is a new language that is designed to rectify some of the weaknesses of Matlab and Python, allowing one to write code that appears syntactically similar to both of them, while also offering high performance.

Julia has reached a critical mass userbase, and it appears that it will be around for the long haul. One of the key facets of Julia is multiple dispatch, a form of polymorphism, that plays a major role in Julia's performance and flexibility.

A downside to Julia, that it shares with Python, is packages for things like plotting, quadrature, interpolation, *etc.* are not in core Julia, but must be added in separately.

C/C++: C and C++ are general purpose systems programming languages that will likely provide the best performance for a given problem. Many scientific computing libraries for PDE, numerical linear algebra, and optimization are written in these languages. Parallel computing is available through MPI and OpenMP.

The downside to C/C++ is that there is a lot more overhead in the code. The same calculation in C/C++ will require more lines of code than would be needed for Matlab/Python/Julia. It is easy to find problems which can be quickly coded and efficiently solved in Matlab/Python/Julia, but would require dozens of lines of code in C/C++ while offering minimal performance gain.

Visualization is typically not performed in C/C++. Instead, output data must be imported in another language or environment (Matlab, Python, Julia, Paraview, Ovito), and then plotted there.

For complicated multiphysics 3D PDE simulations, there are few options other than C/C++.

This course will be taught in Julia, with an emphasis on:

- Splitting problems into modules and scripts. Scripts are for making the actual computation you want, while modules provide the infrastructure needed for said computation. The division between these two can be ambiguous, but it is good to try to make such a distinction. It is overkill for small computations which only require calling a few lines of code.
- Using multiple dispatch. We may want to write methods in our modules to take different types of inputs, but make the morally same computation. Rather than write a single function with a bunch of if/then/else statements, we will make use of typing and multiple dispatch so that the correct "version" of the function will be called when necessary.

2.2. **Development Environments.** Julia can be downloaded at `https://julialang.org/downloads/`. This provides you with a REPL (Read-Evaluate-Print-Loop) environment and little else. It is recommended that you install a development environment on top of this, and install the neccessary add ons to link it to Julia. Two free, and similar, options are:

- Atom, available at `https://atom.io/`. Then install the JUNO add on.
- Microsoft VS Code, available at `https://code.visualstudio.com/`. Then install the Julia language support extension.

You can optionally install the Jupyter extensions.

Once this is installed, it is recommend that you also install Jupyter, so as to be able to work with notebooks in browsers. This can be done by downloading and installing Anaconda, `https://www.anaconda.com/products/individual`. After installing Anaconda, add the package `jupyter-lab`, with either the Anaconda package manager, or at the command line:

```
$ conda install jupyter-lab
```

This will ask you to install a bunch of other packages too, which you should. After installing this package, open Julia, enter the package manager with the `]` key, and run

```
(@v1.6) pkg> add IJulia
```

This should register Julia with your Jupyter installation. If, for some reason, Julia does not appear as an option inside of Jupyter, you can always call

```
(@v1.6) pkg> build IJulia
```

The Jupyter lab server can be launched at the command line with

```
$ jupyter-lab
```

2.3. **Packages.** Once you have Julia installed and working, it is recommended you have the following packages, amongst others, installed: `Plots, PyPlot, Random, Distributed, Distributions, Dierckx, Optim, FFTW, LinearAlgebra, Arpack, SparseArrays, SharedArrays`.

2.4. **GitHub.** You will want to get in the habit of using version control for larger projects. This tracks how your code evolves, and it also facilitates sharing, between machines and with other users/developers. `git` is one of the more popular version control tools, and it will be emphasized in this course. You should make an account at `https://github.com/`. This will allow to create an unlimited number of private repositories with limited sharing, which will be sufficient for this course.

2.5. **Creating a Julia Package with Version Control.**

2.5.1. *Manual Creation.* A suggested way of creating a module that can be the basis of a larger computation is done in the following steps:

(1) Create a (private) git repo on GitHub.
(2) Clone the repo locally.
(3) Generate the basic files for a Julia package; this is a bit awkward.
(4) Work with the local repo, committing and pushing changes.

To create the initial repo, navigate to the GitHub website, login, and click "New." The recommended repository naming convention is `MyPackage.jl`, where `MyPackage` is whatever is an appropriate name for your project. You will generally want to create a private repo (it can be made public later), add a `README`, an `.gitignore`, and a license (MIT is fine). **Note**, it is the convention on GitHub to name Julia packages with the `.jl` extension, but it can be omitted and it will work fine.

After having created the repo, you should be at the webiste for the package, download it to your local machine.

Launch Julia, enter the package manager, and run

```
(@v1.6) pkg> generate MyPackage
```

This will generate a new folder, `MyPackage`. Copy the contents of this new folder over to the `MyPackage.jl` folder that you got from GitHub. Add the `Project.toml` and the `src/MyPackage.jl` files to the repo, commit, and push.

Then, launch Julia inside the `MyPackage.jl` folder, enter the package manager, and create an environment for developing this package:

```
(@v1.6) pkg> activate .
```

Make sure to regularly commit and push changes at logical intervals.

It will eventually be beneficial to add unit testing. When you want to add unit testing, add the folder and file `test/runtests.jl` within the `MyPackage.jl` folder. Inside of the `MyPackage` environment, you will want to create a new project manifest for testing, to add some flexibility,

```
(@v1.6) pkg> activate ./test/
(@v1.6) pkg> add Test
```

and then add any additional modules that you might need, just for testing. The purpose of this is that there may be modules that you need for testing that you do not need for the package itself, and this allows you to separate out these two needs.

2.5.2. *Using Templates.* Another handle tool is found in the `PkgTemplates` which automates a lot of the creation and connection with GitHub. Before using this, make sure you have git installed, and set, in the shell

```
$ git config --global user.name "Hubert Farnsworth"
$ git config --global user.email "hubert.farnsworth@planetexpress.com"
$ git config --global github.user "hubertfarnsworth"
```

This will facilitate the GitHub interaction. Next, assuming you have added the `PkgTemplates` module, run:

```
julia> using PkgTemplates
julia> tpl = Template(;dir="/Users/hjf3000/code/modules/", plugins=[
    Git(;branch="main")])
julia> tpl("MyNewModule")
```

This will create the `MyNewModule` module in the `/Users/hjf3000/code/modules/` folder; it will be populated with all the standard files. If you now go to GitHub and create a totally blank project named `MyNewModule.jl` (public or private), you will be able to do a push from what you have on your local machine, and it will work as though you had done all the preceding steps by hand.

2.6. **Working with Custom Packages.** For packages that are not in the general registry (i.e. your packages), there are several ways of working with them so that they behave like regular packages (i.e. you can call `using MyPackage`).

The first method is to navigate to the directory containing the package, presuming the key files are in the `src` subdirectory, and activate an environment in that folder:

```
(@v1.6) pkg> activate .
```

The second method is to modify the `LOAD_PATH` variable to include your package folder:

```
julia> push!(LOAD_PATH,"path/to/my/package")
```

The third method, for packages that you have hosted on GitHub, is to add it:

```
(@v1.6) pkg> add https://github.com/myusername/MyPackage.jl
```

This last method is particularly useful if you are continuing to develop the package, as, when you do a Julia update, it will alos update this package. This works with both public and private repositories.

3. Two Point Boundary Value Problems

In this section, we will develop our coding skills by solving the two point, divergence form, Dirichlet boundary value problem

e:bvp1d
$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u = f(x), \quad a < x < b \tag{3.1}$$

with the boundary conditions

e:bcs1ddir
$$u(a) = u_L, \quad u(b) = u_R \tag{3.2}$$

for (left and right). Here, the "physical" problem data is the domain, (a, b), the boundary conditions, u_L and u_R, and the terms in the equation, $p(x)$, $q(x)$, and $f(x)$. We then solve for u.

Subject to modest assumptions (i.e., p and q being bounded non-negative measurable functions, with $p(x) \geqslant p_0 > 0$, and using standard methods from functional analysis (i.e., Riesz representation theorem and the Lax-Milgrim, see Theorem 1.57, Lemma 1.59 of [2]), it is a standard result to obtain a unique solution in the Sobolev space $H^1(a, b)$ (see Theorems 2.10 and 2.42 of [2]). This will also generalize to the higher dimensional analog of (3.1).

3.1. Finite Difference Method.

Before addressing (3.1), we consider the simplified problem

e:dirichlet1d
$$-u'' = f(x), \quad a < x < b \tag{3.3a}$$
$$u(a) = u(b) = 0 \tag{3.3b}$$

where we have assumed $p = 1$, $q = 0$, and the boundary conditions are homogeneous.

The first way to solve (3.3), not covered in [2] (but which is discussed in, for instance, [3]) is with finite differences. There are some disadvantages to this approach (i.e., it does not have an elegant functional analytic framework), but:

- Finite differences are a classical, and practical, way to quickly solve many problems;
- Finite differences are simple to understand;

- This will help us build up our software architecture intuition.

The basic idea of the finite difference method is to first introduce a mesh, or lattice, for the domain (a, b),

e:mesh1d (3.4) $$a = x_0 < x_1 < \ldots < x_{n-1} < x_n < x_{n+1} = b$$

Often, the spacing between mesh points is uniform, with

e:mesh1ddx (3.5) $$\Delta x = x_{j+1} - x_j = \frac{b-a}{n+1}.$$

We will approximate $u(x)$ at the mesh points, with a vector $\mathbf{u} = (u_i)$, where $u_i \approx u(x_i)$. From the boundary conditions, we already have $u_0 = u(x_0) = 0$ and $u_n = u(x_n) = 0$, so we need to solve for the other n values.

The idea is now to use Taylor series expansions to replace the second derivative operator (acting on twice differentiable functions) with a lattice operator (acting on $\mathbb{R}^{n-1}$). We begin with a first derivative, where, for sufficiently fine (uniform) mesh spacing

(3.6) $$\begin{aligned} u'(x_i) \approx \frac{u(x_{i+1}) - u(x_i)}{x_{i+1} - x_i} &= u'(x_i) + u''(x_i)(x_{i+1} - x_i) + \mathrm{O}((x_{i+1} - x_i)^2) \\ &= u'(x_i) + u''(x_i)\Delta x + \mathrm{O}(\Delta x^2) \end{aligned}$$

Consequently, a $\mathrm{O}(\Delta x)$ approximation of $u'(x_i)$ is

(3.7) $$\frac{u(x_{i+1}) - u(x_i)}{\Delta x} = u'(x_i) + \mathrm{O}(\Delta x)$$

A similar computation shows that

e:seconddiff (3.8) $$\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{\Delta x^2} = u''(x_i) + \mathrm{O}(\Delta x^2)$$

Thus, formally, for (3.3)

(3.9) $$-\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{\Delta x^2} = f(x_i) + \mathrm{O}(\Delta x^2)$$

This suggests that we can approximate the solution with the linear system

(3.10) $$-\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} = f_i, \quad i = 1, \ldots, n, \quad u_0 = 0, \quad u_{n+1} = 0.$$

with $f_i = f(x_i)$. This can be written in matrix vector form as

e:tridiag1 (3.11) $$\underbrace{-\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & 1 & -2 \end{pmatrix}}_{A} \mathbf{u} = \mathbf{b}$$

3.2. **Implementation in Julia.** To solve this problem, with a uniform mesh, it is sufficient to specify Δx and $\mathbf{b}$ (we will return to generalizations later). Even here, there are a few options to consider that will let us explore multiple dispatch in Julia:

- We can either specify Δx directly, or we can specify n, since $(b-a)/(n+1) = \Delta x$
- We can specify a vector $\mathbf{f}$ of values, or we can specify a function f, from which we can build up the vector $\mathbf{b}$. However, to do this, we will need to know the actual mesh points. These can be obtained if we have either n or Δx **and** a and b.

3.2.1. *Module Design.* In this first example, we will build our underlying problem data structure so as to allow for flexibility in how the problem data is inputted. The core of our solution to this problem will be the package `BasicBVP1D`. We will build this package up to eventually be able to handle non-homogeneous boundary conditions. Our problem will be split up into the following three steps:

Problem setup: Set the values of a, b, n, etc.
Problem assembly: Construct the matrix A and the vector f, if necessary
Problem solution: Compute the vector u

While this is rather pedantic for problem (3.3), it highlights a way of thinking about splitting up our problems into the following phases:

Load Data: This involves setting parameters, loading data from disk, etc.
Preprocessing: This involves any computations necessary after the data is loaded in, but before we can begin what is the "main" part of the problem
Solution: This is the main computational step of the problem
Postprocessing: This is where data is saved to disk, plots are generated, etc.

Our first version of the module will include the following pieces, which we will now fill in:

```
module BasicBVP1D

# data structure encapsulating the problem
struct FiniteDifferenceBVPProblem
...
end

# convenience function for constructing the problem
function FiniteDifferenceBVPProblem(...)
...
end

# assemble the system (matrix and right hand side)
function assemble_system!(...)
...
end

# solve the problem
function solve_bvp(...)
...
```

```
end

export FiniteDifferenceBVPProblem, assemble_system!, solve_bvp

end # end module
```

In the above bit of code, the ellipses, ..., refer to things which must be filled in. The `export` command at the end are included such that we have direct access to these functions:

```
julia> using BasicBVP1D
julia> FiniteDifferenceBVPProblem(...)
```

If we did not export these functions, we would need to do

```
julia> using BasicBVP1D
julia> BasicBVP1D.FiniteDifferenceBVPProblem(...)
```

The first item, the `struct`, is the data structure that will encapsulate everything we need for our problem, along with additional information that may be convenient to have available. In our first version, the problem data structure will be:

```
struct FiniteDifferenceBVPProblem
    a # (a,b) define the domain
    b
    Δx # mesh spacing
    n # number of points in the interior, excluding a and b
    x # interior mesh
    f # value of f at the interior nodes
    A # the matrix
    rhs # right hand side of our system
end
```

Remark 3.1. *Note, there is no* **right way** *to design the problem data structure. Not everything in the above structure is required to solve the problem. Some may argue that we should have the full mesh,* `x`*, including* x_0 *and* x_{n+1}*.*

Now, to conveniently populate the structure we use a **constructor**,

```
function FiniteDifferenceBVPProblem(a, b, n, f)
    Δx = (b-a)/(n+1);
    x = LinRange(a, b, n+2)[2:end-1];

    # allocate the matrix
    A = zeros(n, n);
    rhs = zeros(n);

    return FiniteDifferenceBVPProblem(a, b, Δx, n, x, f, A, rhs)
end
```

If we call this function now, having passed in values of `a, b, n, f`, it will return a data structure containing everything we need:

```
julia> a = 0;
julia> b = 1;
julia> n = 9;
julia> f = ones(n); # elementary choice of f
julia> problem = FiniteDifferenceBVPProblem(a,b,n,f);
```

Next, we need to assemble matrix `A` and vector `rhs`, to describe system (3.11). That is handled by the `assemble_system!` command:

```
function assemble_system!(problem)

    # build first row
    problem.A[1,1] = 2/problem.Δx^2;
    problem.A[1,2] = -1/problem.Δx^2;
    # build rows 2-n-1
    for i in 2:problem.n-1
        problem.A[i,i-1] = -1/problem.Δx^2;
        problem.A[i,i] = 2/problem.Δx^2;
        problem.A[i,i+1] = -1/problem.Δx^2;
    end
    # build row n
    problem.A[problem.n,problem.n-1] =  -1/problem.Δx^2;
    problem.A[problem.n,problem.n] =  2/problem.Δx^2;

    # build rhs, copying over values
    @. problem.rhs = problem.f;

    problem
end
```

Note that this function modifies the argument `problem`. While it is not required, the convention in Julia is that functions which modify their arguments have a `!` in their name, and rather than `return` anything, they have the modified variable(s) at the end. The `@.` term is a macro which is equivalent to the for loop:

```
for i in 1:length(problem.rhs)
    problem.rhs[i] = problem.f[i];
end
```

The two vectors must be the same size, or an error will be generated. Having coded this in, we can then call

```
julia> assemble_system!(problem);
```

Lastly, we implement a solve function,

```
function solve_bvp(problem)
    u = problem.A\problem.rhs;
    return u
end
```

and we can then call

```
julia> u = solve_bvp(problem);
```

3.2.2. *Test Problems.* You can verify that the above code executes without error; but is it right? For this we need a **test problem**. A test problem is a problem for which we have one of the following:

- Know an analytic or semi-analytic solution against which we can compare;
- Have an already verified numerical method that can produce a high resolution solution for comparison.

For (3.3), it is quite easy to generate test problems. Take any sufficiently smooth function satisfying the boundary conditions, and apply $-d^2/dx^2$ to it to get f.

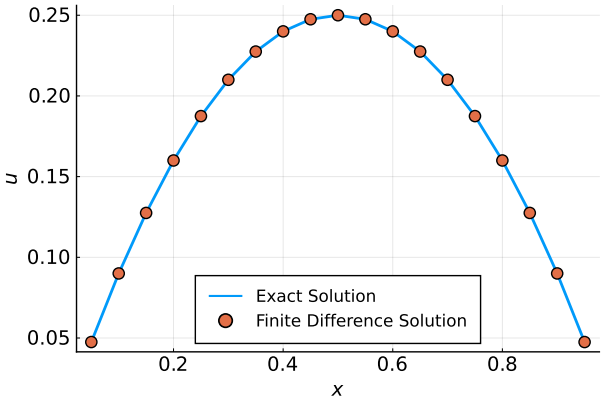

FIGURE 1. Finite difference solution of the boundary value problem $-u'' = 2$ on $(0,1)$ with $u(0) = u(1)$.

fig:parabol1

e:testproblem1

Exercise 3.2. *Verify that $u = x(1-x)$ solves $-u'' = 2$ on $(0,1)$ with boundary conditions $u(0) = u(1) = 0$.*

We will use the test problem form Exercise 3.2 to verify our code:

```
using BasicBVP1D
n = 19;
f = 2*ones(n)
problem = FiniteDifferenceBVPProblem(0, 1, n, f);
assemble_system!(problem);
u = solve_bvp(problem);
```

This is visualized in Figure 1, where see excellent agreement. Indeed, we can quantify the accuracy on the mesh points with the commands:

```
using LinearAlgebra
err = @. problem.x * (1-problem.x) - u;
norm(err, Inf)
```

and this reports an ℓ^∞ norm error of size 1.9e-16. This is within floating point accuracy. Indeed, we have stumbled upon a test problem for which the finite difference solution is exact on the mesh.

Exercise 3.3. *Verify that the finite difference solution to $-u'' = 2$, $u(0) = u(1) = 0$ is exact on the nodes.*

Note that we had to add the `LinearAlgebra` module to gain access to `norm`, and we specified $p = \infty$ with `Inf`. By default, it computes the 2-norm.

That this is an exact solution suggests we should keep it in mind as a test problem. Indeed, we will add it to the test suite for our module. Adding tests to our package means that whenever we change our code, we can verify it still works properly by running the tests. When we deploy the module on a new computer, it also lets us ensure the module behaves as expected. As an example, open Julia, and enter the package manager, then run:

```
(@v1.6) pkg> test Random
```

You will see Julia then perform a sequence of tests; all should pass.

To add a test suite to our module, first, add the folder `test` to the package folder, so that it is at the same level as `src`. Then, inside of `test`, add the file `runtests.jl`. Now, within the package folder, add the `Test` package,

```
(BasicBVP1D) pkg> add Test
```

Then, switch project environments and add the modules we need for testing:

```
(BasicBVP1D) pkg> activate ./test/
(test) pkg> add Test
(test) pkg> add LinearAlgebra
```

These commands will create/modify `Project.toml` files, which you should commit to the repository, along with `runtests.jl`.

Next, we will start creating a test. A basic `runtests.jl` file would be:

```
using BasicBVP1D
using Test
using LinearAlgebra

@testset "Dense" begin
    @test include("dense_test1.jl")
end
```

This sets us up to run a collection of tests, first for the dense matrix formulation of the problem. The first test, is contained in `dense_tests1.jl`, a separate file:

```
# solve the problem
a = 0;
b = 1;
n = 9;
f = 2*ones(n)
problem = FiniteDifferenceBVPProblem(a, b, n, f);
assemble_system!(problem);
u = solve_bvp(problem);

# computer error
x = LinRange(0,1,n+2);
err = @. problem.x * (1-problem.x) - u;
# check error
norm(err, Inf)< 1e-14
```

Remark 3.4. *Note that the last computation that is made is to compute a boolean variable (*`True/False`*). It is these boolean variables that tell us if hte code executed correctly.*

The `dense_test1.jl` file should also be in the `test` folder. If we now run (having switched environments):

```
(BasicBVP1D) pkg> test
```

we will see, at the end:

```
     Testing Running tests...
Test Summary: | Pass  Total
Dense         |    1      1
     Testing BasicBVP1D tests passed
```

3.2.3. *Typing and Multiple Dispatch.* Currently, our code requires us to specify a vector `f` of values that represent the function f. Suppose, instead, we had a function $f(x)$ coded up, and we just wanted to use that, instead of first computing the vector `f`. We can add this functionality in through **typing** and **multiple dispatch** (a form of polymorphism).

First, we modify our problem data structure to allow us to detect what the type of `f` is:

```
struct FiniteDifferenceBVPProblem{TF}
    a # (a,b) define the domain
    b
    Δx # mesh spacing
    n # number of points in the interior, excluding a and b
    x # interior mesh
    f::TF # value of f at the interior nodes
    A # the matrix
    rhs # right hand side of our system
end
```

Note that we have included a `TF` (for the type of `f`). If we now run the code:

```
n = 19;
f = x-> 2;
problem = FiniteDifferenceBVPProblem(0, 1, n, f);
```

we have passed our function in instead of the vector. If we ran `assemble_system!(problem);` on this, we would generate an error, because, currently, the code expects that `f` is a vector, not a function. We need to further modify our code so that it can recognize that it should do something different when `f` is a function. First, we modify our existing `assemble_matrix!` every so slighty, changing the declaration to be:

```
function assemble_system!(problem::FiniteDifferenceBVPProblem{TF})
    where {TF<:AbstractArray}
...
end
```

This instructs Julia that:

- problem is of type `FiniteDifferenceBVPProblem`
- This version of it has `f` of subtype (`<:` means subtype) `AbstractArray`.

We do not need to further modify the existing function. We now write a second verison of the function that will handle the case that `f` is a function:

```
function assemble_system!(problem::FiniteDifferenceBVPProblem{TF})
    where {TF<:Function}

    # build first row
    problem.A[1,1] = 2/problem.Δx^2;
    problem.A[1,2] = -1/problem.Δx^2;
    # build rows 2-n-1
    for i in 2:problem.n-1
        problem.A[i,i-1] = -1/problem.Δx^2;
        problem.A[i,i] = 2/problem.Δx^2;
        problem.A[i,i+1] = -1/problem.Δx^2;
    end
    # build row n
```

```
    problem.A[problem.n,problem.n-1] =  -1/problem.Δx^2;
    problem.A[problem.n,problem.n] =  2/problem.Δx^2;

    # build rhs, evaluating on the nodes
    @. problem.rhs = problem.f(problem.x);

    problem
end
```

The purpose of multiple dispatch is that, at run time, Julia figures out which version of the function to use based on what has been passed in. What makes Julia work so well is that it precompiles different versions of the functions, and then uses the right one for the right data type.

As a simple example of multiple dispatch, consider the action of adding two numbers. To make the idea explicit, rather than use the `+` symbol, we have the function `add`, taking two arguments. There are many different types that we might wish to add together, including integers, floating point numbers, both with different levels of precisions. So, behind the scenes, Julia is really pre-compiling

```
function add(a::Integer, b::Integer)
...
end

function add(a::Integer, b::Float64)
...
end

function add(a::Float64, b::Integer)
...
end

function add(a::Float64, b::Float64)
...
end
```

where each of these is slightly different, but written in an optimal way for each case.

Having added the second version of `assemble_system!`, we will now be able to apply it to our probelm and have it execute properly:

```
n = 19;
f = x-> 2;
problem = FiniteDifferenceBVPProblem(0, 1, n, f);
u = solve_bvp(problem);
```

This executes without error. We should also create a second test file, `dense_test2.jl`, which has `f` as a function instead of a vector. After coding that, we will then have:

```
@testset "Dense" begin
    @test include("dense_test1.jl")
    @test include("dense_test2.jl")
end
```

and when we run the test suite, we will see it has two successful tests.

3.2.4. *Assessing Convergence.* A full proof of convergence of the finite difference method for (3.1) can be found in Chapter 4 of [1]. Here, we will empircally verify that

e:bvperr1

$$\max_j |u_j - u(x_j)| \propto \Delta x^2. \tag{3.12}$$

To demonstrate, this, we will pick a test problem for which there is an error (unlike the one in Exercise 3.2), compute the errors at a sequence of Δx values, and, graphically, infer the rate of convergence.

Exercise 3.5. *Verify that if $u = \sin(\pi x)$, then, taking $f = \pi^2 \sin(\pi x)$, u solves* (3.3) *on* $(0, 1)$*, and that, in a finite difference discretization, there is a non-trivial error.*

How should the Δx be selected? The standard strategy is to plot the errors on a $\log - \log$ scale, so, to be meaningful, the Δx values must be spaced out uniformly on the log scale. This can be accomplished by selecting the Δx's of the form $\Delta x^{(k)} = 2^{-k}$, $\Delta x^{(k)} = 10^{-k}$, or something similar. Recalling that for the Dirichlet problem, $\Delta x = (b-a)/(n+1)$, a reasonable choice, for this problem, is to take

$$n = [5, 10, 20, 40, 80, 160, 320, 640] - 1.$$

With this choice, each n halves the mesh spacing, and the Δx are (relatively) simple ratios of whole numbers. This can be accomplished with the following bit of code

```
f = x-> π^2 * sin(π * x);
u_exact = x-> sin(π * x);
a = 0;
b = 1;
n_vals = [5, 10, 20, 40, 80, 160, 320, 640] .-1;

err_vals = [];
for n in n_vals
    problem = FiniteDifferenceBVPProblem(0, 1, n, f);
    assemble_system!(problem);
    u = solve_bvp(problem);
    err = norm(u .- u_exact.(problem.x), Inf);
    push!(err_vals, err);
end
```

The results of this are plotted in Figure 2. As the figure shows, the error appears to be $\propto \Delta x^2$.

To understand a bit of why the error is quadratic, a sketch of the analysis is as follows. From (3.8), we expect that if we insert the exact solution into $A\mathbf{u} = \mathbf{b}$, we will find

$$(Au)(x_j) = -\frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1})}{\Delta x^2} = f(x_j) + \mathrm{O}(\Delta x^2)$$

Meanwhile finite difference solution exactly satisfies $(A\mathbf{u})_j = f(x_j)$. Consequently, using that A is invertible,

$$\begin{aligned} |(u(x_j)) - \mathbf{u}\|_\infty &= |A^{-1}(A(u(x_j)) - A\mathbf{u})|_\infty \\ &\leqslant \|A^{-1}\|_\infty \|A(u(x_j)) - A\mathbf{u})|_\infty \\ &\lesssim \|A^{-1}\|_\infty \Delta x^2 \end{aligned}$$

The two steps that must be completed to make this fully rigorous are:

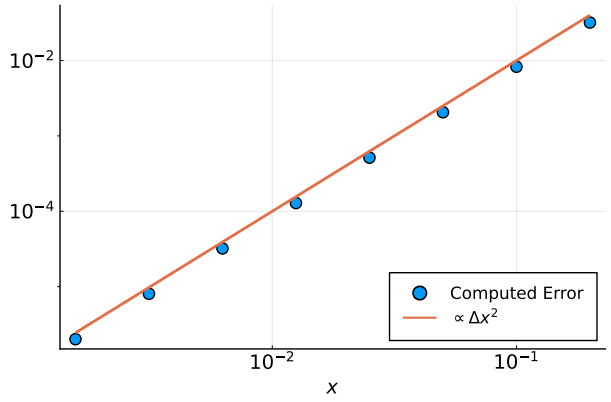

FIGURE 2. Error of the finite difference method for (3.3) using the test problem $f = \pi^2 \sin(\pi x)^2$.

fig:finite_diff_err

- Determine the implicit constant in the $O(\Delta x^2)$ error of (3.8) and verify that it is uniform over all Δx sufficiently small.
- Verify that the matrix $\|A^{-1}\|_\infty$ is uniformly bounded over all Δx sufficiently small.

Both of these things will turn out to be true, but it is beyond the scope of this course.

3.2.5. *Benchmarking Performance.* To check the performance of our code, we use the `BenchmarkTools` package, and we can then run, on any function or code snippet:

```
julia> using BenchmarkTools
julia> @btime randn();
```

and observe that this reports the time, along with the number of times and amount of memory that is allocated. We will now check this in our code to see how things scale with the mesh size/number of mesh points:

```
f = x-> π^2 * sin(π * x);
a = 0;
b = 1;
n_vals = [5, 10, 20, 40, 80, 160, 320, 640] .-1;

for n in n_vals
    problem = FiniteDifferenceBVPProblem(0, 1, n, f);
    assemble_system!(problem);
    @btime u = solve_bvp($problem);
end
```

Observe that this only tets the time to solve the linear system, we can repeat this exercise to check the time it takes to construct and assemble the system. The dollar sign is needed by the `@btime macro`, to treat the argument as fixed.

You should observe that on the solve command, each doubling of the mesh appears to quadruple the memory usage, and the time increases by factor between two and four.

If we test the system assembly performance, we see the time and memory approximately doubles with each doubling of the mesh.

If we test the construction of the problem, we will see timing scale by a factor of two to four with each mesh doubling, and memory usage go up by a factor of four with each doubling.

3.2.6. *Exploiting Sparsity.* Recall that the matrix in (3.11) is sparse, which is to say, most of the entries are zeros. This motivates us to try using `SparseArrays`, which allows for the representation and use of sparse matrices. To implement this, we will need to add this to the project and create a second constructor which constructs a sparse representation of the matrix. This amounts to creating a new constructor, `SparseFiniteDifferenceBVPProblem`, and altering the matrix construction line to be:

```
A = spzeros(n, n);
```

If we now revisit the benchmarking on this, we find the solver, has both linear timing and memory usage, and, once the system is big enough, far better performance than in the dense case. System assembly is comparable for both dense and sparse problems, while the construction is now only about linear in n.

3.2.7. *Interpolation.* Recall that we obtain a solution on the mesh, $\{x_j\}$. Suppose, having computed $\mathbf{u}$, the approximation at the mesh, we wish to estimate the value at another value of x that fails to be a mesh point. We can do this by interpolation. Let us denote the interpolation operator, $\mathcal{I}$.

This can be accomplished using one of the interpolation packages, including `Dierckx` and `Interpolations` amongst others. For example, assuming we have already solved (3.3), we could construct piecewise linear interpolant with the commands

```
u = solve_bvp(problem);
u_itp = LinearInterpolation([0; problem.x; 1], [0; u; 0]);
```

We can then evaluate `u_itp` on any other values in the domain, even if they are not mesh points. We alternatively could have made this construction with `CubicSplineInterpolation`, which is a piecewise cubic spline interpolant.

Which interpolant should be used? Here, we turn to the issue of a balance of errors. Since the standard interpolants (lineaer and cubic) are linear transformations of the vector of values,

$$\begin{aligned} |u(x) - \mathcal{I}\mathbf{u}(x)| &\leqslant |u(x) - \mathcal{I}(u(x_j))(x)| + |\mathcal{I}(u(x_j))(x) - \mathcal{I}\mathbf{u}(x)| \\ &\lesssim \underbrace{|u(x) - \mathcal{I}(u(x_j))(x)|}_{\text{Interpolation Error}} + \underbrace{\|(u(x_j)) - \mathbf{u}\|_\infty}_{\text{Finite Difference Error}} \\ &\lesssim |u(x) - \mathcal{I}(u(x_j))(x)| + \Delta x^2 \end{aligned} \tag{3.13}$$

Recall (see, for instance, [4, 5]), that for sufficiently smooth functions u:

$$\text{Piecewise Linear Interpolant: } |u(x) - \mathcal{I}(u(x_j))(x)| \lesssim \Delta x^2 \tag{3.14}$$

$$\text{Cubic Spline Interpolant: } |u(x) - \mathcal{I}(u(x_j))(x)| \lesssim \Delta x^4 \tag{3.15}$$

(3.16)

For this problem, there is no sense on using the higher order cubic interpolant, because the error will be dominated by the quadratic finite difference error. Hence, the sensible choice is to use a piecewise linear interpolant here.

3.3. **Functional Analysis Framework.**

3.4. **Finite Elements.**

3.5. **Spectral Methods.**

References

larsson2003partial
[1] S. Larsson and V. Thomée. *Partial differential equations with numerical methods*, volume 45. Springer, 2003.

lord2014introduction
[2] G. J. Lord, C. E. Powell, and T. Shardlow. *An introduction to computational stochastic PDEs*, volume 50. Cambridge University Press, 2014.

morton2005numerical
[3] K. W. Morton and D. F. Mayers. *Numerical solution of partial differential equations: an introduction*. Cambridge university press, 2005.

Plato.2002
[4] R. Plato. *Concise Numerical Mathematics*. AMS, 2002.

suli2003introduction
[5] E. Süli and D. F. Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003.