Elements of Scientific Computing with Julia

March 5, 2015

Scientific Computing

Scientific computing is the process of coming up with algorithms and computer programs to solve quantitative and qualitative problems arising in areas such as the engineering and behavioral sciences, as well as biology and finance.

Scientific Computing

Scientific computing is the process of coming up with algorithms and computer programs to solve quantitative and qualitative problems arising in areas such as the engineering and behavioral sciences, as well as biology and finance.

A lot of interesting problems in these areas do not have exact or analytic solutions. Scientists must rely on numerical methods for approximating these solutions.

Math + CS

Scientific Computing techniques draw from mathematics and computer science:

1. Math will give us the theory and the ability to come up with the appropriate models and numerical techniques for solving a problem; as to which element is in a given spot in a set.
2. CS will give us the algorithmic and programming tools for putting our models to work efficiently.

What we'll cover:

1. Numerical Integration
2. Stencil Computation
3. Numerical Linear Algebra
4. Numerical Optimization
5. Linear Regression
6. Logistic Regression
7. And More..

What we'll cover:

1. Numerical Integration
2. Stencil Computation
3. Numerical Linear Algebra
4. Numerical Optimization
5. Linear Regression
6. Logistic Regression
7. And More..

Notice that these are not the conventional topics covered in a more stereotypical scientific computing course. My goal is for this course to serve as a general overview of computational methods from scientific computing, statistical computing and machine learning.

Good Computational Scientists care about...

the programming language they should use: it needs to be high performing and flexible, while not being painful to use.

Good Computational Scientists care about...

the programming language they should use: it needs to be high performing and flexible, while not being painful to use.

In this course we will use Julia, a high performing and dynamic programming language. It's syntax is friendly and flexible for programming scientific computing algorithms.

Good Computational Scientists care about...

the programming language they should use: it needs to be high performing and flexible, while not being painful to use.

In this course we will use Julia, a high performing and dynamic programming language. It's syntax is friendly and flexible for programming scientific computing algorithms.

More information on Julia can be found at julialang.org

Good Computational Scientists care about...

presentation, collaboration and revision control.

Good Computational Scientists care about...

presentation, collaboration and revision control.

Scientists must make sense of their mathematical models and computational results, then present their results in a clean, standard and modular format via the use of tools such as LaTeX (for typesetting documents) and `git` (for revision control).

Our computational scientist's tool box

1. *Mathematical tools*: will be acquired through the lectures, homeworks, required and optional readings.

Our computational scientist's tool box

1. *Mathematical tools*: will be acquired through the lectures, homeworks, required and optional readings.
2. *Julia*: a high performing technical computing programming language.

Our computational scientist's tool box

1. *Mathematical tools*: will be acquired through the lectures, homeworks, required and optional readings.
2. *Julia*: a high performing technical computing programming language.
3. *Git*: a revision control and source code management system focused on speed, widely used for sharing source code and keeping track of different versions of computer programming projects.

Our computational scientist's tool box

1. *Mathematical tools*: will be acquired through the lectures, homeworks, required and optional readings.
2. *Julia*: a high performing technical computing programming language.
3. *Git*: a revision control and source code management system focused on speed, widely used for sharing source code and keeping track of different versions of computer programming projects.
4. *GitHub*: a web-based hosting service for computer programming projects using the Git RCS.

Our computational scientist's tool box

1. *Mathematical tools*: will be acquired through the lectures, homeworks, required and optional readings.
2. *Julia*: a high performing technical computing programming language.
3. *Git*: a revision control and source code management system focused on speed, widely used for sharing source code and keeping track of different versions of computer programming projects.
4. *GitHub*: a web-based hosting service for computer programming projects using the Git RCS.
5. LaTeX: a document markup language that uses the TeX typesetting program for formatting text output, it is widely used in academia and for presenting mathematical formulas on various websites.

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem:

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem:

1. usage of simplified model (since there are many complexities in real world problems that are difficult to account for);

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem:

1. usage of simplified model (since there are many complexities in real world problems that are difficult to account for);
2. our measurement equipment have finite precision or we were not able to gather that much data anyway;

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem:

1. usage of simplified model (since there are many complexities in real world problems that are difficult to account for);
2. our measurement equipment have finite precision or we were not able to gather that much data anyway;
3. we may have to discretize a continuous problem in order to solve it;

Good enough?

Scientific computing is all about approximations, and approximations produce errors!

The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem:

1. usage of simplified model (since there are many complexities in real world problems that are difficult to account for);
2. our measurement equipment have finite precision or we were not able to gather that much data anyway;
3. we may have to discretize a continuous problem in order to solve it;
4. truncation and rounding will happen due to computer precision.

Error Analysis

We must study the effect of approximations and computational errors on our results in order to better understand our results. Such study is called *error analysis*.

Example

Consider computing the surface area of the Earth by using the formula: $A = 4\pi r^2$.

- This formula gives the surface area of a perfect sphere, which is not necessarily the true shape of the Earth;

Example

Consider computing the surface area of the Earth by using the formula: $A = 4\pi r^2$.

- This formula gives the surface area of a perfect sphere, which is not necessarily the true shape of the Earth;
- The radius of the Earth is based on a combination of measurements taken by equipment of finite precision;

Example

Consider computing the surface area of the Earth by using the formula: $A = 4\pi r^2$.

- This formula gives the surface area of a perfect sphere, which is not necessarily the true shape of the Earth;
- The radius of the Earth is based on a combination of measurements taken by equipment of finite precision;
- π is a transcendental infinite quantity, but it will have to be truncated at some point;

Example

Consider computing the surface area of the Earth by using the formula: $A = 4\pi r^2$.

- This formula gives the surface area of a perfect sphere, which is not necessarily the true shape of the Earth;
- The radius of the Earth is based on a combination of measurements taken by equipment of finite precision;
- π is a transcendental infinite quantity, but it will have to be truncated at some point;
- The final result will be rounded by our computers depending on their precision.

Total Error

A scientific computing problem can be view as computing the value of a function $f : \mathbf{R} \to \mathbf{R}$.

Total Error

A scientific computing problem can be view as computing the value of a function $f : \mathbf{R} \to \mathbf{R}$.

If we denote the true value of the input data by x, the true result by $f(x)$, the inexact input by $\hat{x}$ and the approximated function by $\hat{f}$, then:

Total Error

A scientific computing problem can be view as computing the value of a function $f : \mathbf{R} \to \mathbf{R}$.

If we denote the true value of the input data by x, the true result by $f(x)$, the inexact input by $\hat{x}$ and the approximated function by $\hat{f}$, then:

Total Error $= \hat{f}(\hat{x}) - f(x) = (\hat{f}(\hat{x}) - f(\hat{x})) + (f(\hat{x}) - f(x)) =$ computational error $+$ propagated data error.

Computational versus Data Error

That is, the difference between using the approximate function and the exact function on the same input is a *computational error*.

Computational versus Data Error

That is, the difference between using the approximate function and the exact function on the same input is a *computational error*.

While the difference between using the exact function on inexact input versus exact input is called a pure *data error*.

Computational Error

Computational Error will be either truncation or rounding errors. *Note: data error may have more to do with other biases or limitations in collecting the data, we won't talk much it.*

1. Truncation error is the difference between the result we get after truncating an infinite series (or other infinite quantities) and the result that would be produced if we were to use exact arithmetic.
2. Rounding error is the difference between the result we get due to using finite precision computers and the result that would be produced if we were to use exact arithmetic. Thus, *computational error* is the sum of these two errors.

Absolute versus Relative Error

We can represent errors from numerical computations in either *absolute* or *relative* form:

Absolute versus Relative Error

We can represent errors from numerical computations in either *absolute* or *relative* form:

$$\textit{Absolute error} = \text{approximate value - true value}$$
$$\textit{Relative error} = (\text{approximate value - true value})/\text{true value}$$

Absolute versus Relative Error

We can represent errors from numerical computations in either *absolute* or *relative* form:

$$\begin{aligned} \textit{Absolute error} &= \text{approximate value - true value} \\ \textit{Relative error} &= \text{(approximate value - true value)/true value} \end{aligned}$$

Note that the relative error can be seen as a percentage of the true value, if we multiply it by 100.

Condition Number

Besides analyzing the error of a approximate solution, it is also wise to understand the conditioning of our model.

Condition Number

Besides analyzing the error of a approximate solution, it is also wise to understand the conditioning of our model.

A problem is *well-conditioned* if a relative change in the input date causes a similar relative change in the solution.

Condition Number

Besides analyzing the error of a approximate solution, it is also wise to understand the conditioning of our model.

A problem is *well-conditioned* if a relative change in the input date causes a similar relative change in the solution.

While an *ill-conditioned* problem is one where a relative change in the input data causes a much larger relative change in the solution.

Condition Number

Given these definitions, we can calculate the condition number of a problems as follows (we denote it by $\mathcal{K}$):

$$\mathcal{K} = \frac{|\text{relative change in solution}|}{|\text{relative change in input data}|} = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|}$$

Example

As an illustrative example, consider the problem of computing the values of $y = \cos(x)$ near $\pi/2$.

Example

As an illustrative example, consider the problem of computing the values of $y = \cos(x)$ near $\pi/2$.

Let $x = \pi/2$ and let h be a small perturbation to x. Then we have:

Example

As an illustrative example, consider the problem of computing the values of $y = \cos(x)$ near $\pi/2$.

Let $x = \pi/2$ and let h be a small perturbation to x. Then we have:

$$\textit{Absolute error} = \cos(x+h) - \cos(x) \approx -h\sin(x) \approx -h$$

Example

As an illustrative example, consider the problem of computing the values of $y = \cos(x)$ near $\pi/2$.

Let $x = \pi/2$ and let h be a small perturbation to x. Then we have:

$$\textit{Absolute error} = \cos(x+h) - \cos(x) \approx -h\sin(x) \approx -h$$

This follows by the definition of the derivative:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}.$$

Example cont.

On the other hand we have

Example cont.

On the other hand we have

$$Relative\ error =$$

$$\frac{\cos(x+h)-\cos(x)}{\cos(x)} \approx -h\frac{\sin(x)}{\cos(x)} = -h\tan(x) \approx \infty$$

Example cont.

On the other hand we have

Relative error =

$$\frac{\cos(x+h)-\cos(x)}{\cos(x)} \approx -h\frac{\sin(x)}{\cos(x)} = -h\tan(x) \approx \infty$$

Note that a small change in the input data for $\cos(x)$ near $\pi/2$, causes a large relative change in the output, regardless of how we computed it.

Activity

Let's write a Julia program to compute the mathematical constant e, the base of natural logarithms, from the definition:

$$e = \lim_{n \to \infty} (1 + 1/n)^n$$

Activity

Let's write a Julia program to compute the mathematical constant e, the base of natural logarithms, from the definition:

$$e = \lim_{n \to \infty} (1 + 1/n)^n$$

We will compute $(1 + 1/n)^n$ for $n = 10^k$, $k = 1, 2, ..., 20$. Our goal will be to determine the error in the successive approximations we get by comparing them with the value of the built in constant e.

Activity

Let's write a Julia program to compute the mathematical constant e, the base of natural logarithms, from the definition:

$$e = \lim_{n \to \infty} (1 + 1/n)^n$$

We will compute $(1 + 1/n)^n$ for $n = 10^k$, $k = 1, 2, ..., 20$. Our goal will be to determine the error in the successive approximations we get by comparing them with the value of the built in constant e.

A question for us to think about: do you think the error always decreases as n increases?

Activity

Enter the following code into IJulia:

```
for k = 1:20
    a = (1+1/(10^k))^(10^k)
    err = a - e
    @printf("For k = %d, we get the approximation:
            %1.12f and the error %1.12f \n",
            k, a, err)
end
```

Then discuss your results and conclusions on Canvas.