MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 2: Rootfinding and Interpolation

Lindon Roberts

Due date: 9am, Tuesday 19 April (week 7)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Orders of Convergence

As we have discussed in lectures, a common way of comparing the quality of algorithms is to measure their order of convergence. Recall an algorithm producing values $x_1, x_2, \ldots$ converging to a solution x^* has order m if there exists a constant C (with $C < 1$ if $m = 1$) such that

$$|x_{n+1} - x^*| \leq C|x_n - x^*|^m,$$

for all n sufficiently large. In practice, we try to measure an error at each iteration, $e_1, e_2, \ldots \to 0$ and estimate the observed order of convergence.

Lab Book 1. (a) Given a vector $\boldsymbol{e} = (e_1, \ldots, e_n)$ of errors $e_n > 0$, explain mathematically how fitting a straight line through the data $\boldsymbol{x} = (\log e_1, \ldots, \log e_{n-1})$ and $\boldsymbol{y} = (\log e_2, \ldots, \log e_n)$ can estimate the order of convergence for $e_n \to 0$. (b) Using `np.polyfit` to do the fitting, estimate the order of convergence of the following sequence [5 points]

$$\begin{aligned}
e_1 &= 9.55213728 \times 10^{-1} \\
e_2 &= 9.06436704 \times 10^{-1} \\
e_3 &= 7.33394145 \times 10^{-1} \\
e_4 &= 3.89531478 \times 10^{-1} \\
e_5 &= 5.83960912 \times 10^{-2} \\
e_6 &= 1.97897053 \times 10^{-4} \\
e_7 &= 7.63529928 \times 10^{-12}
\end{aligned}$$

2 Rootfinding

Implement Newton's method and the secant method for solving $f(x) = 0$. You can use the below code as a starting point: note that this code uses some fancy formatting for the `print` command to display the results at each iteration nicely. The only termination condition here is the maximum number of iterations.

```python
def newton(f, df, x0, niters):
    """
    Newton's method for 1D rootfinding.
    - The function f(x) is the one we want the root of
    - The function df(x) is the derivative f'(x)
    - x0 is the starting point
    - niters is the number of iterations to run
    """
    x = x0  # initial guess
    print("{0:^3}{1:^25}{2:^25}".format("k", "xk", "f(xk)"))
    for i in range(niters):
        print("{0:^3}{1:^25.15e}{2:^25.15e}".format(i, x, f(x)))
        ############
        # TODO Add code here to calculate the new value of x
        x = x - 1  # temporary code, replace this
        ############
    return x

def secant(f, x1, x2, niters):
    """
    Secant method for 1D rootfinding.
    - The function f(x) is the one we want the root of
    - x1 and x2 are the two starting points
    - niters is the number of iterations to run
    """
    f1 = f(x1); f2 = f(x2)
    print("{0:^3}{1:^25}{2:^25}".format("k", "xk", "f(xk)"))
    for i in range(niters):
        print("{0:^3}{1:^25.15e}{2:^25.15e}".format(i, x2, f2))
        if f1 == f2:
            print('Secant method error: division by zero')
            return x2
        ############
        # TODO add code here to calculate the new iterate x3
        x3 = x2 - 1  # temporary code, replace this
        ############
        # Update x1 and x2 (don't need to modify this)
        x1 = x2; f1 = f2
        x2 = x3; f2 = f(x3)
    return x2
```

Lab Book 2. Make a plot of the function $f(x) = e^{-x} - x$, and use this to decide on a sensible choice of starting points. Run Newton's method and the secant method for 10 iterations to find a root of $f(x)$. Estimate the order of convergence (e.g. using $f(x_n) \to 0$) and comment all the results you have obtained. [9 points]

Lab Book 3. Run Newton's method to find a root of $f(x) = x^4$ starting from $x = 1$. Estimate the order of convergence and comment on the results. [3 points]

In practice, the best methods for rootfinding in one unknown use a mixture of the bisection method (since it is guaranteed to converge) and a faster method like Newton. In Python, rootfinding methods are part of SciPy's `optimize` module.[1]

```python
import scipy.optimize as optimize

# Example code to find a root of f(x) in the interval [-1,1]
# Using Brent's method, but other algorithms are available (check documentation)
soln = optimize.root_scalar(f, bracket=(-1, 1), method='brentq')
print("Root is x =", soln.root)
```

Lab Book 4. Brent's method does not require the derivative $f'(x)$. For $f(x) = e^{-x} - x$, how does it compare to the secant method? [2 points]

Hint: you may want to look at `soln.iterations` and `soln.function_calls`. Check the documentation to see what these are.

The `scipy.optimize` package has lots of routines for minimizing functions, least-squares data fitting, and multidimensional rootfinding.

2.1 Functions with Multiple Roots

Make a plot of the function

$$f(x) = \cos\left(\frac{1}{x}\right),$$

for $x \in [0.03, 0.2]$. You should see that $f(x)$ has many roots in this interval. What happens to $f(x)$ as $x \to 0$?

Lab Book 5. Try running Newton's method for this $f(x)$ with starting points $x_0 = 0.05, 0.06, \ldots, 0.2$. What do you observe? [4 points]

[1] SciPy is included in Anaconda.

2.2 Multidimensional Rootfinding

Look at the online documentation for `scipy.optimize`, and figure out what functions it has for multidimensional rootfinding (i.e. solving nonlinear systems of equations). Using one of these functions, find a solution $\boldsymbol{x} \in \mathbb{R}^2$ to the nonlinear system[2]

$$\begin{aligned} x_1 + x_2 - x_1 x_2 + 2 &= 0, \\ x_1 e^{-x_2} - 1 &= 0. \end{aligned}$$

Lab Book 6. Implement the multidimensional version of Newton's method from lectures and use it to solve the above system of equations starting from $\boldsymbol{x}_0 = (0.1, -1)$. Do you observe a quadratic local convergence rate? [4 points]

Hint: to measure the error, I suggest using $\sqrt{f_1(\boldsymbol{x}_n)^2 + f_2(\boldsymbol{x}_n)^2}$, where f_i are the two functions you wish to make zero.

Then, use `scipy.optimize` to find a solution to the system

$$\begin{aligned} -13 + x_1 + ((5 - x_2)x_2 - 2)x_2 &= 0, \\ -29 + x_1 + ((x_2 + 1)x_2 - 14)x_2 &= 0. \end{aligned}$$

I recommend trying two different starting points: $\boldsymbol{x}_0 = (6, 6)$ and $\boldsymbol{x}_0 = (10, 0)$. You do not have to submit these results for your lab book.

3 Interpolation

In this section we will find an interpolating polynomial for a function. NumPy has several routines for manipulating polynomials, but here we will use the simplest one, `numpy.poly1d`. To represent the polynomial

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

we provide `numpy.poly1d` with a vector of coefficients in reverse order (i.e. $[a_n, a_{n-1}, \ldots, a_1, a_0]$). For example, to represent $p(x) = 4 - 3x + x^2$ we use

```python
# Define vector of coefficients (in decreasing powers of x)
a = np.array([1.0, -3.0, 4.0])
# Build polynomial object
p = np.poly1d(a)
# Evaluate polynomial at x=12
print(p(12))  # can also input a vector of points to evaluate at
```

Given data $(x_0, y_0), \ldots, (x_n, y_n)$, we can find the coefficients $a_0, \ldots, a_n$ by solving the Vandermonde linear system from lectures.

Lab Book 7. (a) Implement a function `polyfit` which takes in a vector of x values $(x_0, \ldots, x_n)$ and y-values $(y_0, \ldots, y_n)$ and returns the corresponding interpolating polynomial as a NumPy polynomial object (i.e. you should have `return np.poly1d(a)` as the last line of the function). You should do this by solving the corresponding Vandermonde system. (b) Use your function to find the degree-10 polynomial interpolating $\sin(x)$ at $x = 0, 0.6, 1.2, \ldots, 6$. (c) Make a plot of your interpolant and $\sin(x)$ over $x \in [-2, 9]$ and comment on the quality of the fit. [7 points]

[2]Problem source: `https://support.sas.com/documentation/cdl/en/imlug/66112/HTML/default/viewer.htm#imlug_genstatexpls_sect004.htm`.

Warning! When plotting a function and its interpolant, make sure your plot uses many x-values, including points which you did not use to construct the interpolant. Otherwise, it will appear as if your interpolant is always perfect!

3.1 SciPy's Interpolation Routines

As described in the lectures, there are other ways to construct the same interpolating polynomial without solving the Vandermonde linear system: other approaches have better numerical stability and make it easier to update our polynomial with new nodes or data.

For example, the `scipy.interpolate` module gives us an easy way to construct the interpolating polynomial in the Barycentric form:

$$p_n(x) = \frac{\sum_{i=0}^{n} \frac{w_i}{x-x_i} y_i}{\sum_{i=0}^{n} \frac{w_i}{x-x_i}},$$

where

$$w_i = \frac{1}{(x_i - x_0)(x_i - x_1)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n)}, \qquad \text{for } i = 0, \ldots, n.$$

Given a function $f(x)$ which uses NumPy element-wise functions, we can interpolate $f(x)$ at equally spaced points with:

```python
import scipy.interpolate as interpolate

# Select interpolation nodes
x = np.linspace(-1, 1, 6)  # equally spaced nodes
fx = f(x)  # evaluate f at each interpolation node

# Build interpolant for data (x, fx)
# p is a function for our interpolating polynomial; call it with p(x)
p = interpolate.BarycentricInterpolator(x, fx)
```

Lab Book 8. Using 11 equally spaced nodes in $[-1, 1]$, find the polynomial interpolant for Runge's function

$$f(x) = \frac{1}{1 + 25x^2}.$$

Make a graph of the true $f(x)$ and the interpolant $p(x)$ and comment on what you see. [3 points]

We know that equally spaced points are not usually the best choice for polynomial interpolation. Instead, we should use Chebyshev points. The following function template can be used to calculate these points (you need to implement the calculation yourself):

```python
def chebyshev_points(n):
    '''
    Returns a vector of the Chebyshev points for interpolation to polynomials of degree n

    The vector has length n+1.
    '''
    return np.zeros((n+1,))  # TODO complete this function!
```

Lab Book 9. Repeat the above experiment using Chebyshev points (i.e. interpolate Runge's function at $n + 1$ Chebyshev points for $n = 10$). Plot the true $f(x)$ and your new interpolant $p(x)$ and compare to your previous result. [5 points]

Instead, we could interpolate $f(x)$ using splines (piecewise continuous polynomials). Look at the documentation for `scipy.interpolate.CubicSpline` to see how to build a cubic spline interpolant.

Lab Book 10. Repeat the above experiment using 11 equally spaced points in $[-1, 1]$ and a cubic spline with "not-a-knot" extra conditions. Plot $f(x)$ and your new interpolant and compare to your result for polynomial interpolation with equally spaced points. [3 points]

4 Multivariate Interpolation

There are several methods for interpolation to functions of multiple variables, including splines. Another common method is 'radial basis function' (RBF) interpolation. Given nodes $x_0, \dots, x_n$ (these are vectors) and data $y_0, \dots, y_n$, the RBF interpolant is given by

$$RBF(x) = \sum_{i=0}^{n} w_i \phi(\|x - x_i\|).$$

That is, our interpolant is a linear combination of basis functions which depend on how far away we are from each node (and the process of interpolation requires finding the weights w_i). There are many possible choices of the function ϕ; a common example is the Gaussian function $\phi(r) = e^{-r^2}$.

SciPy implements several multivariate interpolation methods, including RBFs.

```python
# Make some example 2D data: zi = f(xi, yi)
x = np.array([0.0, 1.0, -1.0, -1.0, 1.0])
y = np.array([0.0, 1.0, -1.0, 1.0, -1.0])
z = x**2 + x*y + y**3 + 3
# Build an RBF interpolant z=RBF(x,y)
rbf = interpolate.Rbf(x, y, z, function='gaussian')
# Evaluate the RBF at a particular point
print(rbf(0.0, 0.0))
```

Try building RBFs using this data (or create your own). Evaluate the RBF at some points close to your interpolation nodes, some points between your interpolation nodes, and some points far away from your nodes. Have a look at the documentation of the `Rbf` function and try other choices for $\phi(r)$.

If you want, try to figure out how to generate 2D heat map and 3D plots of 2D functions/data (matplotlib can do this) and plot your data and different choices of interpolant.