hw2

Name: Ruffin White
ID: A53229660

February 8, 2018

Contents

HW2 : Math for Robotics

Author: Ruffin White
Course: CSE291
Date: Feb 8 2018

1.

> Prove that the first derivative $p_2'(x)$ of the parabola interpolating $f(x)$ at $x_0 < x_1 < x_2$ is equal to the straight line which takes on the value $f[x_{i1}, x_i]$ at the point $(x_{i1} + x_i)/2$, for $i = 1, 2$.

Given the shorthand notaion for the divided diffrence defines:

$$f[x_{i1}, x_i] = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

And that $p_2(x)$ and $p_2'(x)$ for a parabola can be written as:

$$p_2(x) = ax^2 + bx + c$$
$$p_2'(x) = p_1(x) = 2ax + b$$

From this we can see $p_1(x)$ assumes a line in the form of $y = mx + b$. The general pressmis we'd like to prove can be stated as so:

$$f[x_{i-1}, x_i] = p_1(x) \quad x = (x_{i-1} + x_i)/2 \quad i \in [0, 1, 2]$$

Substituting for x in $p_1(x)$, we get:

$$\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} = a(x_{i-1} + x_i) + b$$

Given that $p_2(x)$ is equal to $f(x)$ for the samples we have:

$$f(x_i) = p_2(x_i), \quad i \in [0, 1, 2]$$

We can substitute $p_2(x)$ for $f(x)$ in the divided diffrence, them simplify:

$$\frac{(ax_i^2 + bx_i + c) - (ax_{i-1}^2 + bx_{i-1} + c)}{x_i - x_{i-1}} = a(x_{i-1} + x_i) + b$$
$$\frac{(ax_i^2 + bx_i) - (ax_{i-1}^2 + bx_{i-1})}{x_i - x_{i-1}} = a(x_{i-1} + x_i) + b$$
$$\frac{a(x_i - x_{i-1})^2 + b(x_i - x_{i-1})}{x_i - x_{i-1}} = a(x_{i-1} + x_i) + b$$
$$a(x_i - x_{i-1}) + b = a(x_{i-1} + x_i) + b$$

So, unless there is an error in my assumptions or reductions, I think there might be a typo in that the point given should be instead: $(x_i - x_{i-1})/2$, for $i = 1, 2$. This would then satisfy the above proof, when substituting the point on the right hand side.

2.

a

> Implement Muller's method.

See Jupyter Notebook for code.

b

Use Muller's method to find all the roots of the polynomial $p(x) = x^3 - 4x^2 + 6x - 4$.

Using our Muller's method, we'll start by seaching for our first root. (Points close to the roots are chosen here to prevent exorbitantly large print-outs in the document for demonstration)

```
  x_i:  2.1
  x_{i-1}:  2.01
  x_{i-2}:  1.9
Iteration:  0
  A:  0.0296018181818178
  B:  0.3946909090909126
  C:  0.401818181818186
  x_i:  2.1
  x_{i-1}:  2.01
  x_{i+1}:  2.00004974877
Iteration:  1
  A:  -0.00233051558625
  B:  0.022091878273
  C:  -1.10008305618e-05
  x_i:  2.00004974877
  x_{i-1}:  2.1
  x_{i+1}:  1.99999997511
Iteration:  2
  A:  5.20528934176e-09
  B:  -9.95966316446e-05
  C:  -4.97985752115e-08
  x_i:  1.99999997511
  x_{i-1}:  2.00004974877
  x_{i+1}:  2.0
Iteration:  3
  A:  1.23884476907e-15
  B:  4.9749025369e-08
  C:  1.2250733729e-13
  x_i:  2.0
  x_{i-1}:  1.99999997511
  x_{i+1}:  2.0
Iteration:  4
  A:  -4.37427537128e-21
  B:  -1.20791972002e-13
  C:  1.77635246511e-15
  x_i:  2.0
  x_{i-1}:  2.0
  x_{i+1}:  2.0
Iteration:  5
  A:  -2.61174078467e-17
  B:  -3.63031969069e-15
  C:  -1.80210114142e-15
  x_i:  2.0
  x_{i-1}:  2.0
  x_{i+1}:  2.0
Iteration:  6
  A:  0.0
  B:  8.881784197e-16
```

```
  C:  0.0
  x_i:  2.0
  x_{i-1}:  2.0
  x_{i+1}:  2.0
root:  2.0
```

Having found $x = 2$ to be our first root, we can can deflate our original polonomal by deviding it:

$$q(x) = \frac{p(x)}{x-2} = \frac{x^3 - 4x^2 + 6x - 4}{x-2}$$

```
quotient:  [ 1. -2.  2.]
remainder:  [ 0.]
```

We find our next Q after aplying the polynomial division:

$$q(x) = x^2 - 2x + 2$$

```
  x_i:  1j
  x_{i-1}:  (1.01+1j)
  x_{i-2}:  1
Iteration:  0
  A:  (1.0097980201979802+1.03019798020198j)
  B:  (4.039596040395961+0.04039596040395984j)
  C:  (3.0096990300969906-0.96990300969903j)
  x_i:  1j
  x_{i-1}:  (1.01+1j)
  x_{i+1}:  (1+1j)
Iteration:  1
  A:  (0.00990099009901+0j)
  B:  (-6.53231808923e-18+0.019801980198j)
  C:  4.3969228698e-18j
  x_i:  (1+1j)
  x_{i-1}:  1j
  x_{i+1}:  (1+1j)
Iteration:  2
  A:  (4.93038065763e-32+0j)
  B:  (4.93038065763e-32-4.4408920985e-16j)
  C:  0j
  x_i:  (1+1j)
  x_{i-1}:  (1+1j)
  x_{i+1}:  (1+1j)
root:  (1+1j)
```

Now we have found $x = 1 + 1j$ as our next root, and because a complex root always has a complex conjugate, we know $x = 1 - 1j$ to be our other root as well, thus all three roots have been found. Just to check our work, we can test this against numpy's own root finding function, by providing it the same coefficients for $p(x)$ as see this matchs.

```
numpy_roots:  [ 2.+0.j  1.+1.j  1.-1.j]
```

3.

Suppose you wish to build an interpolation table with entries of the form $(x, f(x))$ for the function $f(x) = sin(x)$ over the interval $[0, \pi]$. Please use uniform spacing between points.

- How fine must the table spacing be in order to ensure 6 decimal digit accuracy, assuming that you will use linear interpolation between adjacent points in the table?
- How fine must it be if you will use quadratic interpolation?
- In each case, how many entries do you need in the table?

For determinting the greatest possible error using linear interpolation, we'll quantify the error using this theorem:

Let f(x) be twice continuously differentiable on an intercal $[a, b]$ which contains the points $\{x_0, x_1\}$. Then for $a \leq x \leq b$,

$$f(x) - p(x) = \frac{(x - x_0)(x - x_1)}{2} f''(x_0)$$

For our example, we let $f(x) = sin(x)$; we take $0 \leq x, x_0, x_1 \leq 10$. For definiteness, let $x_0 < x_1$ with $h = x_1 - x_0$. Then

$$f''(x) = -sin(x)$$

$$sin(x) - p(x) = \frac{(x - x_0)(x - x_1)}{2} [-sin(x)]$$

As we are interpolating with $x_0 \leq x \leq x_1$, we have

$$(x - x_0)(x - x_1) \geq 0, \quad x_0 \leq c_x \leq x_1$$

Therefore

$$\frac{(x - x_0)(x - x_1)}{2} [-sin(x)] \leq \frac{(x - x_0)(x - x_1)}{2} \max |-sin(x)| \leq \frac{(x - x_0)(x - x_1)}{2} (1)$$

By simple geometry or calculus

$$\max_{x_o \leq x \leq x_1} (x - x_0)(x - x_1) \leq \frac{h^2}{4}$$

Therefore

$$0 \leq sin(x) - p(x) \leq \frac{h^2}{8}$$

So we must solve for an n such that h satisfies

$$0 \leq sin(x) - p(x) \leq \frac{h^2}{8} \leq 10^{-7}$$

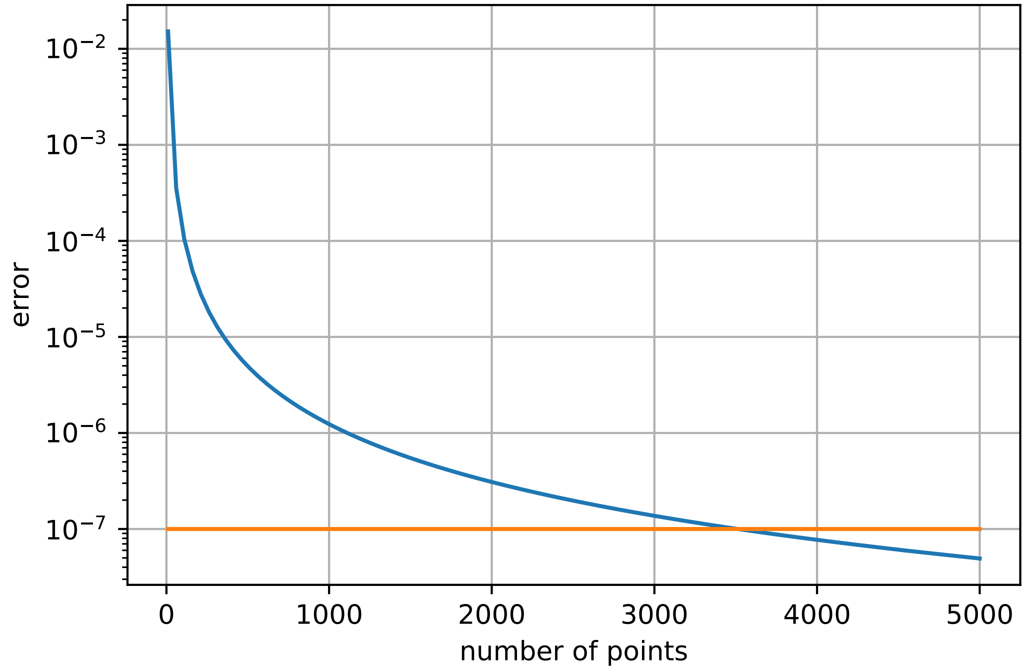
Given that $\frac{\pi - 0}{n} = h$

$$\frac{\left(\frac{\pi - 0}{n}\right)^2}{8} \leq 10^{-7}$$

Thus

$$n \geq 500\sqrt{5}\pi \approx 3512.41$$

We can double check this theoretical calculation by comparing it to an quantitative evaluation, simply by sweeping n through $[10, 50000]$ and collecting the maximum error sampled from the interpolation and plotting the results.



Checking the plot, we see this matches our initial estimate. Additionally, this quantitative evaluation remain useful for when the function f is poorly defined or has no know derivative. If f is simple to compute over the range of interest, this still provided us a good estimate in determining the minimal number of samples to table to satisfy a given error margin.

For determinting the greatest possible error using quadratic interpolation, we'll quantify the error using this general case theorem:

> Let f be a function in $C^{n+1}[a,b]$, and let P be a polynomial of degree $\leq n$ that interpolates the function f at $n+1$ distinct points $x_0, x_1, \ldots, x_n \in [a,b]$. Then to each $x \in [a,b]$ there exists a point $\xi \in [a,b]$ such that

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x - x_i)$$

So, the greatest possible error is bounded by the maximal value of the right hand side:

$$\left| \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x - x_i) \right|$$

But, given n in this case means the size of the polinomail used in interpolation, for the quadratic case, $n = 2$:

$$\left| \frac{1}{3!} f^{3}(\xi_x) \prod_{i=0}^{2} (x - x_i) \right|$$

We'll assume a maximum interval between points of $[0, \pi]$, such that for $x_0, x_1, \ldots, x_n, \xi_x \in [0, \pi]$, then we can determine:

$$\max_{x_0 \leq x \leq x_1} |(x - x_0)(x - x_1)(x - x_2)| = \frac{2h^3}{3\sqrt{3}}$$

Plus, given the nth derivative of sin is either $\pm cos$ or $\pm sin$, we know:

$$\max \left| f^{(n+1)}(\xi_x) \right| = 1$$

So to achive an error less than 10^{-7}, we can solve for:

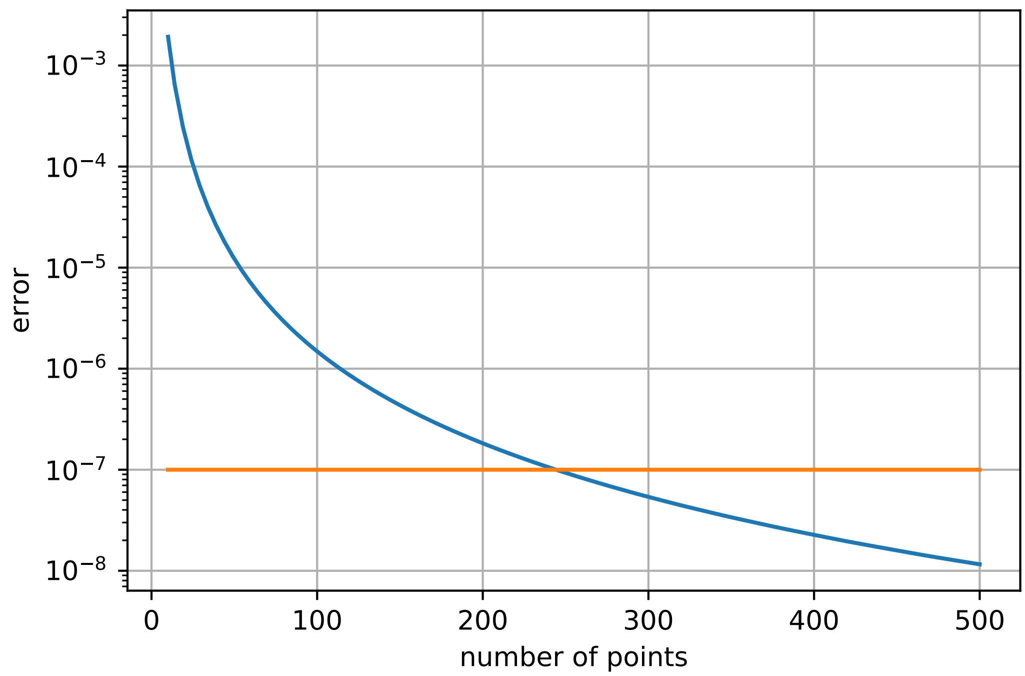
$$\frac{h^3}{9\sqrt{3}}(1) \leq 10^{-7}$$

Given that $\frac{\pi - 0}{n} = h$, where n is now the number of samples

$$\frac{\left(\frac{\pi-0}{n}\right)^3}{9\sqrt{3}}(1) \leq 10^{-7}$$

Thus

$$n \geq \frac{\pi 100 \sqrt[3]{10}}{3^{5/6}} \approx 270.946$$



Again, we find our theoretical and quantitative results coincide quite closely, our theoretical estimate perhaps a bit more conservative. Note the order of magnitude number of lesser points needed to achieve the same error threshold for quadratic as compared to linear. This just goes to show the tradeoffs in model complexity and points needed to accurately interpolate our trigonometric function.

4.

Implement Newton's Method. Consider the following equation:

$$x = tan(x)$$

There are an infinite number of solutions x to this equation. Use Newton's method (and any techniques you need to start Newton in regions of convergence) to find the two solutions that are closest to 5.

Given we know that $tan(x) = \frac{sin(x)}{cos(x)}$, we can see that $f(x) = tan(x) - x$ will be undifined for periodicity of $k \cdot \pi + \frac{\pi}{2}, k \in \mathbb{N}$. Additionally, we can decern the direvitive to be $f'(x) = sec^2(x) = \frac{1}{cos^2(x)}$.

Thus we can use the three closest singulatiesties to bracket the two intervals containing the two closest roots to 5. We can see the two closet intervals to be $[0 \cdot \pi + \frac{\pi}{2}, 1 \cdot \pi + \frac{\pi}{2}]$, and $[1 \cdot \pi + \frac{\pi}{2}, 2 \cdot \pi + \frac{\pi}{2}]$.

For this task, we will narrow down the region of convergance by using bisection. This is gaernted to get us arbitrarily close to the root for regions where the function is continious. This is helpful for trigmaetric functions that may be non-monotonic, as we can provide Newton's Method an accurate innital guess closer to the root to help ensure convergance.

We'll also adjust our initial bounds inward into the region by a factor of a small α, to help ensure our starting points are nuericaly stable. E.g. $x1 = 0 \cdot \pi + \frac{\pi}{2} + \alpha$ and $x2 = 1 \cdot \pi + \frac{\pi}{2} - \alpha$

```
alpha:  1e-09
x1:  1.5707963277948966
x2:  4.71238897938469
f(x1):  -999999980.063
f(x2):  999999728.85


Bisection:  0
  xmid:  3.141592653589793
  fmid:  -3.14159265359
Bisection:  1
  xmid:  3.9269908164872414
  fmid:  -2.92699081749
Bisection:  2
  xmid:  4.319689897935966
  fmid:  -1.90547634068
Bisection:  3
  xmid:  4.516039438660328
  fmid:  0.511300030476
Bisection:  4
  xmid:  4.417864668298147
  fmid:  -1.121306469
Bisection:  5
  xmid:  4.466952053479238
  fmid:  -0.474728284
Bisection:  6
  xmid:  4.491495746069783
  fmid:  -0.038293539557
Bisection:  7
  xmid:  4.5037675923650555
  fmid:  0.219861715301
Bisection:  8
  xmid:  4.497631669217419
  fmid:  0.0869803711173
```

We can see the bisection method has zeroed down quite quickly to an intial estiment that is within a braket only 10^{-3} wide in only a few iterations.

```
Iteration:  0
  x0:  4.49333404098
```

```
  dx:  0.00152218125895
Iteration:  1
  x0:  4.49340592209
  dx:  7.13894724633e-05
Iteration:  2
  x0:  4.4934092911
  dx:  3.3679276088e-06
Iteration:  3
  x0:  4.49340945004
  dx:  1.58931849015e-07
Iteration:  4
  x0:  4.49340945754
  dx:  7.50006901029e-09
x0_newtom:  4.49340945754
```

To compare our simple newtons methods above, we can check agianst numpy's own newton and Brent's (1973) method, where Brent's method is generally considered the best of the rootfinding routines in numpy, as it combines root bracketing, interval bisection, and inverse quadratic interpolation.

```
x0_numpy_newton:  4.49340945754
```

```
x0_numpy_brentq:  4.493409457909064
```

We find all three methods return identical roots. Now lets check the second closest root.

```
alpha:  1e-09
x1:  4.71238898138469
x2:  7.853981632974483
f(x1):  -1000000105.67
f(x2):  999999603.244


Bisection:  0
  xmid:  6.283185307179586
  fmid:  -6.28318530718
Bisection:  1
  xmid:  7.0685834700770345
  fmid:  -6.06858347108
Bisection:  2
  xmid:  7.461282551525759
  fmid:  -5.04706899427
Bisection:  3
  xmid:  7.657632092250121
  fmid:  -2.63029262311
Bisection:  4
  xmid:  7.755806862612302
  fmid:  2.39736342742
Bisection:  5
  xmid:  7.7067194774312116
  fmid:  -0.965267114109
Bisection:  6
  xmid:  7.731263170021757
```

```
  fmid:  0.376522572133
Bisection:  7
  xmid:  7.718991323726484
  fmid:  -0.35610373287
Bisection:  8
  xmid:  7.72512724687412
  fmid:  -0.00742820523835


Iteration:  0
  x0:  7.72524989965
  dx:  0.000115614559031
Iteration:  1
  x0:  7.72525180504
  dx:  1.90368615183e-06
Iteration:  2
  x0:  7.72525183641
  dx:  3.13723473866e-08
Iteration:  3
  x0:  7.72525183693
  dx:  5.16990894539e-10
x0_newtom:  7.72525183693


x0_numpy_newton:  7.72525183693


x0_numpy_brentq:  7.725251836937708
```

Again, we find similar roots with all three methods, demonstrating the proper function of our bisection and newton's method implementations. Had our bisection method failed, then we may have needed to make our α more conservative by shrinking it further to ensure less of the region between deschutes is overlooked. This may also in turn have required the increase in numerical precision, such as switching to double length types to accommodate the more extreme starting point values nearer the disjoints.