Lecture 08: Basins of attraction

Announcement and Intro

Geoff is away for two weeks on parental leave. In the meantime you've got me - Eric. I'm covering the course for those two weeks - lectures and tutorials.

A bit of background:

I'm starting my second year of my Applied Maths PhD with Geoff. This is my first proper lecturing, so I'll try my best.

I was in your position not long ago. I learnt to code Python for my honours in 2016.

I understand what it's like to have no clue what's going on, **so please ask questions** . If you don't want to ask me, ask your neighbour, both of you benefit when one person explains.

What I do is fluid dynamics simulations. Specifically, simulating fluid-solid interactions. More specifically, I use dedalus (http://dedalus-project.org/), which uses "spectral methods" to solve PDEs. It was developed by Geoff and some collaborators, and it's awesome.

In that time, I've learned Python from scratch. I hope I can help you learn it too.

A fun example of what I've learned to do:

In [ ]:

```
from IPython.lib.display import VimeoVideo
VimeoVideo('243498999',width=700,height=400)
```

This is a simulation of 2D flow past an inclined obstacle with blue streaklines (essentially, patterns of dye released from the left side). You can clearly see the formation of vortices shedding off the side of the oval. *So pretty*! There's more like it at the dedalus vimeo (https://vimeo.com/dedalus) webpage.

What's the point?

I'm very excited to help communicate what I like about this course. Computational mathematics shows you how we harness one of the most important technologies of the 20th century - computers - to accomplish incredibly cool things.

And aside from being fascinating, it's **useful**. Abstract theory is great, but theoretical advances are most useful when we can put them into practice - using computers.

A final note: there is a lifetime's worth of information about using Python (and everything else) out on the web. Learn to use it!

Improving jupyter notebooks

Some seriously useful things to improve the Jupyter notebooks

Nbextensions

- Collapsible headings
- Table of contents
- Toggle all line numbers
- Preview markdown
- ...

Command mode shortcuts

- `h`: Help for shortcuts
- `j`: move down a cell
- `k`: move up a cell
- `b`: insert new cell below
- `x`: cut cell
- `c`: copy cell
- `v`: paste cell below
- `y`: convert cell to code
- `m`: convert to markdown
- ...

Recap

We've seen how to solve simple equations using Newton's method. In this lecture, we'll explore some of the ways that process can fail. And also some of the interesting consequences of this failure.

I'll assume you understand the usefulness of object oriented programming from last week. Especially for plotting! For another fun explanation, see this blog post (https://inventwithpython.com/blog/2014/12/02/why-is-object-oriented-programming-useful-with-a-role-playing-game-example/) about using OOP for designing a better RPG.

Stability

A process or algorithm can be either "*stable*" or "*unstable*". In the case of Newton iteration, stable implies the algorithm converges to the desired results. And this remains the case if we make a small change in the initial guess. Unstable can mean a lot of different things. In the words of Leo Tolstoy in the first line of Anna Karenina:

```
"All happy families are alike. All unhappy families are unhappy in their o
wn way."
```

This sentiment applies to stability (happy families), and instability (unhappy families). But the analogy is not a value judgment. There are just a lot more ways something can be unstable:

- it could run away to ***infinity***;
- it could oscillate ***periodically*** around one or more solutions;
- it could wander around ***chaotically***;
- or it could display extreme ***sensitivity to initial conditions*** (the butterfly effect).

The appreciation of stability and instability is one of the biggest and most important differences between engineers and theoretical scientists. For reasons of safety and efficiency, engineers usually want to find ways to keep a system stable under the influence of noise and uncertainty. Theoretical scientists usually become excited when they discover a new way for a system to become unstable. This implies new nonlinear phenomena to explore and understand.

Before, we were builders. This week, we are explorers.

Basins of attraction

The first important thing to consider is the *BASIN OF ATTRACTION*. This is very much the same idea as a "Continental Divide"

Any kind of iteration works the same way. Where you go depends on where you start. ***This is not only the case for Newton iteration.*** It's a very general concept in dynamical systems. But Newton's method is a great place to explore what's going on.

Example 0

$$f(x) = (x-1)(x-3), \quad F(x) = x - \frac{f(x)}{f'(x)} = \frac{x^2-3}{2(x-2)}$$

Notice that $x = 2$ causes a problem for $F(x)$. This should be a sign that something funny happens on either side.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:

```
# One line function syntax
def f(x): return (x-1)*(x-3)
def F(x): return 0.5*(x**2 - 3)/(x-2)
# Anonymous, or lambda functions
f = lambda x: (x-1)*(x-3)
F = lambda x: 0.5*(x**2 - 3)/(x-2)
```

In [3]:

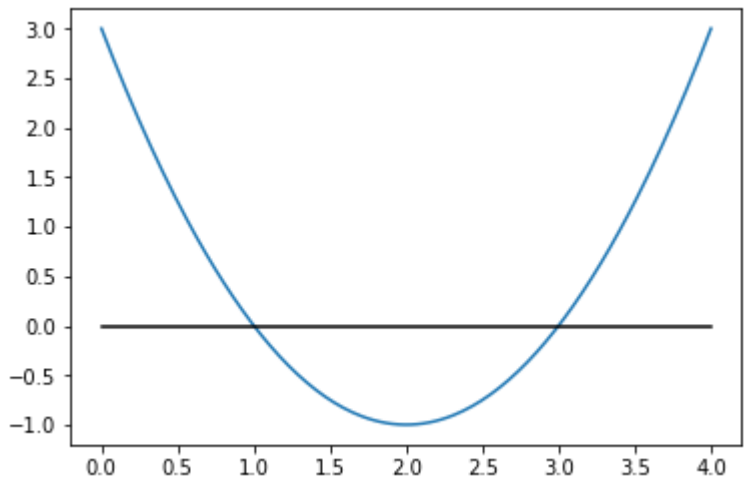
```
x = np.linspace(0,4,200)
```

In [4]:

```
# Good OOP plotting
fig, ax = plt.subplots()
ax.plot(x,f(x))
ax.plot(x,0*x,color='black')
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7f78f556bcf8>]
```



The function has two roots at $x = 1, 3$. The function also has two basins of attraction: $x < 2$ and $x > 2$. Any initial condition with $x_0 < 2$ will converge to the $x = 1$ root, and vice versa.

The boundary between the basins ($x_0 = 2$) gives undefined results. Locations where $f'(x) = 0$ will cause problems for Newton's method. But understanding how is instructive for understanding how things become difficult in other situations.

What happens for $x_0 \approx 2$?

While loops

To experiment with this, we'll introduce a new Python control structure. The "`while` loop." You need to know what you are doing if you use this loop. It's easy to get in and never leave.

In [5]:

```
# This should converge to either 1, or 3 depending on where you start.

tol, imax, x0 = 1e-10, 50, 1

xold    = 1.99999
eps, i  = np.inf, 0

print(' i        eps      |f(x)|      x ')
print('--    -------    --------   ------')
while (eps > tol) and (i < imax):

    xnew = F(xold)
    eps  = abs(xnew-xold)
    xold = xnew
    i += 1 # same as i = i + 1

    # old printing
#     print('%2i    %.1e    %.1e    %.2f' %(i,eps,abs(f(xnew)),xnew))
    # better printing
    print('{:>2d}   {:>8.1e}    {:>7.1e}  {:>9.2f}'.format(i,eps,abs(f(xnew)),xn
ew))

if (i==imax): print(i+1,'Warning: reached max iterations.')
```

```
 i        eps      |f(x)|      x
--    -------    --------   ------
 1    5.0e+04    2.5e+09  -49998.00
 2    2.5e+04    6.2e+08  -24998.00
 3    1.2e+04    1.6e+08  -12498.00
 4    6.2e+03    3.9e+07   -6248.00
 5    3.1e+03    9.8e+06   -3123.00
 6    1.6e+03    2.4e+06   -1560.50
 7    7.8e+02    6.1e+05    -779.25
 8    3.9e+02    1.5e+05    -388.63
 9    2.0e+02    3.8e+04    -193.31
10    9.8e+01    9.5e+03     -95.66
11    4.9e+01    2.4e+03     -46.83
12    2.4e+01    6.0e+02     -22.43
13    1.2e+01    1.5e+02     -10.23
14    6.1e+00    3.7e+01      -4.16
15    3.0e+00    9.0e+00      -1.16
16    1.4e+00    2.0e+00       0.26
17    5.8e-01    3.4e-01       0.84
18    1.5e-01    2.1e-02       0.99
19    1.1e-02    1.1e-04       1.00
20    5.6e-05    3.1e-09       1.00
21    1.6e-09    0.0e+00       1.00
22    0.0e+00    0.0e+00       1.00
```

String formatting

Python 3 has a new syntax for formatting strings. Some people don't like it because it's different to the old way.

I (Eric) think it's better. The syntax is simple

```
'{:paw.st}'.format(thing)
```

where:

- p: padding character. If omitted, pad with spaces
- a: alignment of string in field
  - <: left align
  - ^: centre align
  - >: right align
- w: width of the field in characters
- .s: digits after decimal point
- t: type of format:
  - `e`: exponential `1.2e+03`
  - `f`: float `1.23`
  - `d`: integer `1`

There's a bunch of other manipulations too. There's heaps of documentation online.

DANGER - Critical points

Remember, Newton's method shoots to the next point along the tangent to the curve. This makes points close to critical points go a long way before they come back. In this simple case, there is nothing else going on far away. That is not the case with many other functions.

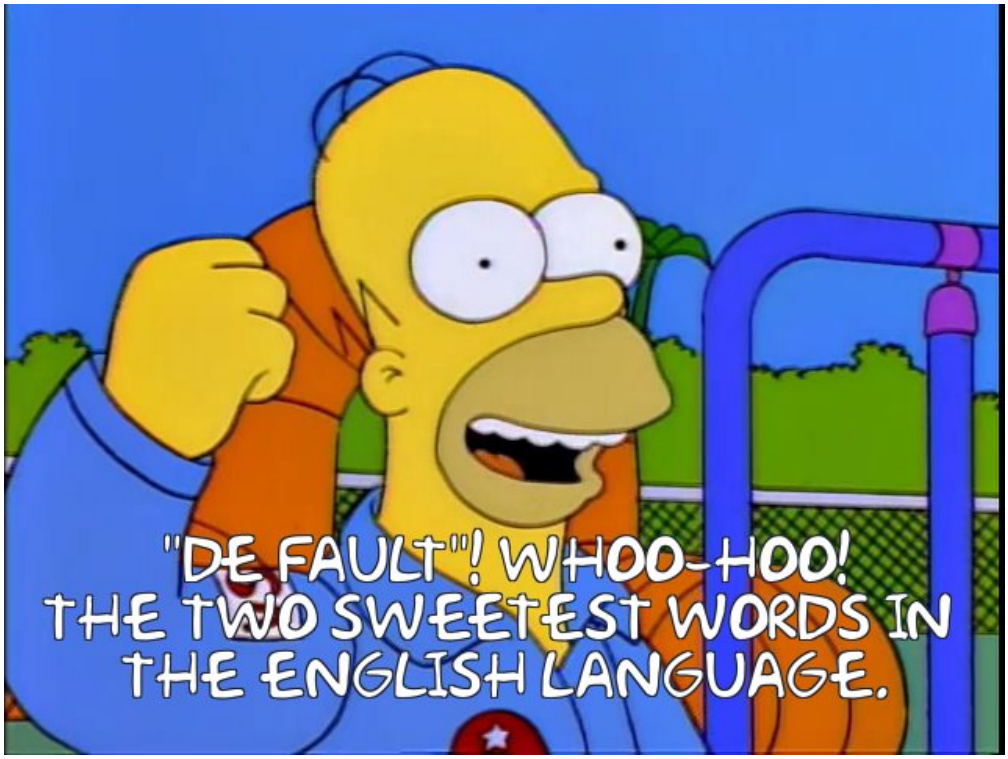
Example 1

$$f(x) = x^4 - 5x^3 - 50x^2 + a$$

The following code defines this function. It shows a little more Python features.

Default arguments

It's possible to define "*optional*" arguments with a sensible defaults:



This is useful if you don't want to have to specify an argument all the time. It's also useful because keyword arguments don't have to be in order. Learn to use them well.

Below, we use one positional argument `x`, and two keyword arguments, `a` (the constant term) and `deriv` (optional return of derivative)

```
def f(x,a=248,deriv=False):
```

In [6]:

```
def f(x,a=248,deriv=False):

    c = [a,0,-50,-5,1]

    fun = 0
    for i in range(len(c)): fun += c[i]*(x**i)

    if (not deriv): return fun

    der = 0
    for i in range(1,len(c)): der += i*c[i]*(x**(i-1))

    return fun, der
```

We can see how to use non-default values in a function, and also the same with the `pyplot.plot` function.

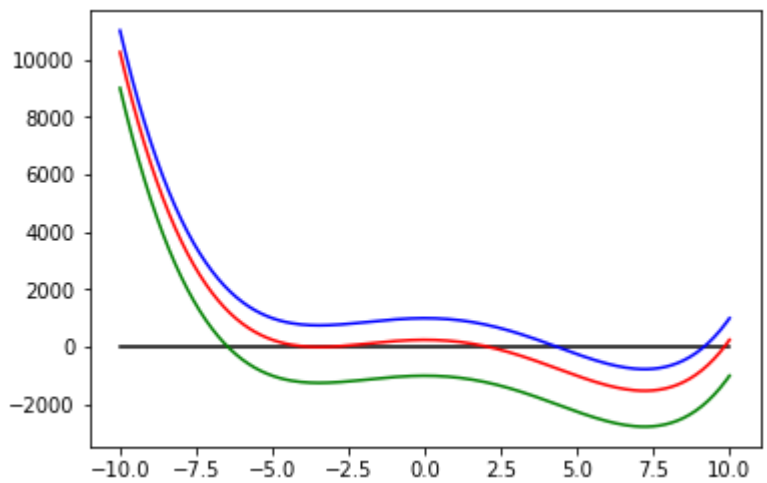
You can see the function has a different number of real roots depending on the value of a. This will effect the stability of Newton's method in interesting ways.

In [7]:

```
x = np.linspace(-10,10,200)   # Make a useful domain.
fig, ax = plt.subplots()
ax.plot(x,0*x,color='black') # Show the location of the horizontal axis.

# This uses the default value; a=248
ax.plot(x,f(x),color='red')

# These use other values
ax.plot(x,f(x,a= 1000),color='blue')
ax.plot(x,f(x,a=-1000),color='green')
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x7f78f54a3710>]
```



Simultaneous assignment

Let's look at the derivative. Because of the way we wrote it, we need to get the function at the same time. The Python syntax of putting multiple definitions on one line is helpful in many situations; e.g,

```
a, b, c = 1, 2, 3
```

However, it's not always possible; so you need to be careful. Everything needs to be defined, and Python needs to be able to "unpack" the results.

In [8]:

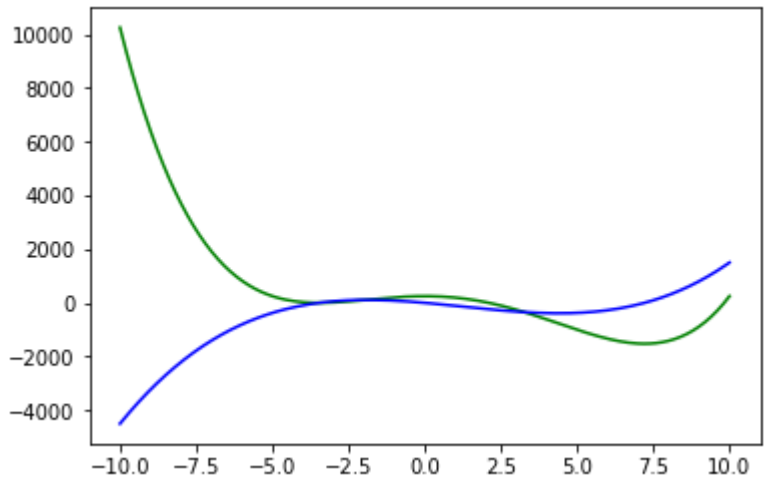
```
y, Dy = f(x,deriv=True)
```

In [9]:

```
fig, ax = plt.subplots()
ax.plot(x,y,color='green')
ax.plot(x,Dy,color='blue')
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x7f78f5417dd8>]
```



In [10]:

```
def basin_map(a=248,niter=40,npoints =100,domain=(-10,10)):

    # unpack the doamin values
    x0, x1 = domain[0], domain[1]

    # create an array of zeros of the right shape to fill with values.
    x = np.zeros([niter, npoints])

    # initialise the iteration with equally spaced values over the doamin
    x[0]=np.linspace(x0,x1,npoints)

    # run the Newton iteration
    for i in range(1,niter):
        y, Dy = f(x[i-1],a=a,deriv=True)
        x[i]  = x[i-1] - y/Dy

    # return the array of updated values.
    return x
```

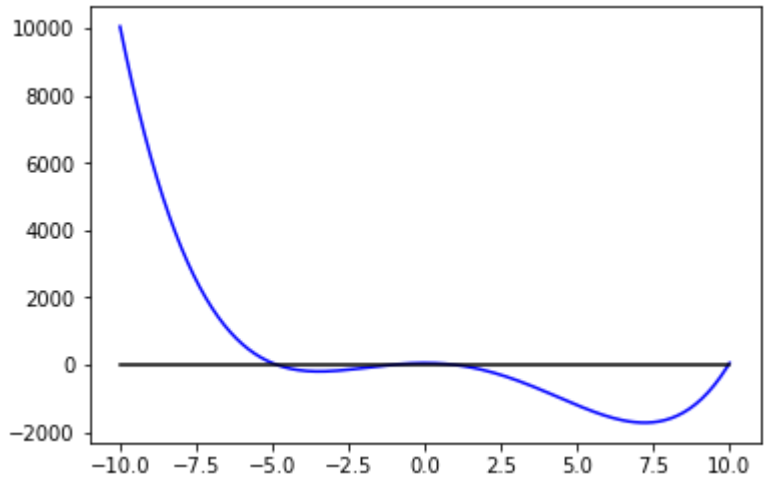
This makes a picture based on which starting points go to which roots.

In [11]:

```
x = np.linspace(-10,10,200)
fig, ax = plt.subplots()
ax.plot(x,f(x,a=50),color='blue')
ax.plot(x,0*x,color='black')
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x7f78f538d6a0>]
```

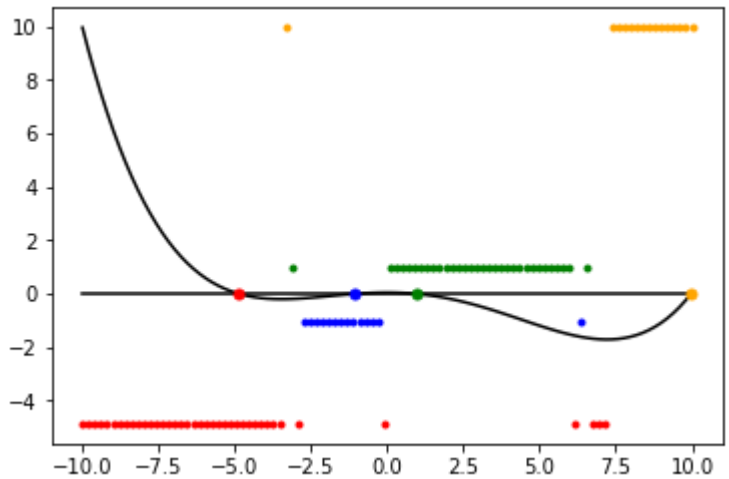


Try changing the default to other values, e.g., `a=-1000`, `a=300`, or `a=1000`.

In [12]:

```
a = 50
x = basin_map(a=a)

tol = 1e-10

xsrt  = x[0]
xend  = x[-1]
xend  = np.round(xend/tol)*tol # remove everything after 10th digit.

roots = np.unique(xend)
print(roots)

colors=['red','blue','green','orange','purple','black','grey','yellow']

# Plot rescaled function
fig, ax = plt.subplots()
ax.plot(xsrt,max(roots)*f(xsrt,a=a)/max(f(xsrt,a=a)),color='black')

# Plot x-axis
ax.plot(xsrt,0*xsrt,color='black')

# Plot basins of attraction
kmax = min(len(roots),len(colors))
for k in range(kmax):
    r = (xend == roots[k]) # find where each root starts.

    # Plot the kth root
    ax.plot(roots[k],0,'.',color=colors[k],markersize=10)

    # Plot the basis of attraction of the kth root
    ax.plot(xsrt[r],xend[r],marker=".",linestyle="",color=colors[k])
```

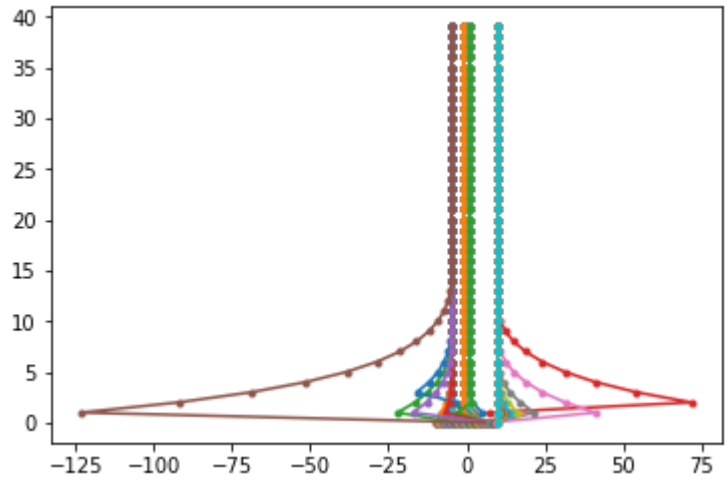
```
[-4.85736478 -1.07224753  0.96324642  9.96636589]
```



This shows how long it takes for everything to settle down; `x.T` takes the "transpose" of the 2D array.

In [13]:

```
for i in range(len(x.T)):
    plt.plot(x[:,i],range(40),'.-')
```



Why Newton's method works:

Stability of fixed points

First consider *any* iteration; not just a Newton iteration.

$$x_{n+1} = F(x_n)$$

Suppose there is a *fixed-point* value a such that

$$a = F(a)$$

We can analyse the stability of the iteration by assuming we are close to the value a and see what happens from there.

Therefore assume

$$x_n = a + \delta_n$$

where δ_n is small. Use Taylor series

$$a + \delta_{n+1} = F(a + \delta_n) = F(a) + \delta_n F'(a) + \frac{\delta_n^2}{2} F''(a) + \ldots$$

Or

$$\delta_{n+1} = \delta_n F'(a) + \frac{\delta_n^2}{2} F''(a) + \ldots$$

If $F'(a) \neq 0$, then the ***stability*** of the iteration will depend on if $|F'(a)| < 1$. If $|F'(a)| > 1$ then a small error will run away from the fixed point.

Newton's method stability

For Newton's method

$$F(x) = x - \frac{f(x)}{f'(x)}$$

It's simple to prove that $F'(a) = 0$, if $f(a) = 0$. This implies that

$$\delta_{n+1} = \frac{f''(a)}{2f'(a)} \delta_n^2 + \dots$$

This is why the number of digits in the error of Newton's method doubles every iteration. And you can see that everything works *provided* $f'(a) \neq 0$. And that it's not too small either. If $f'(a) \approx 0$, then the method will converge *provided* you start close enough. But it may be very difficult to get close enough.

Halley's method stability

Halley's method replaces $f(x) \to f(x)/\sqrt{|f'(x)|}$. This makes the second-order term vanish.

For Halley's method:

$$\delta_{n+1} = \frac{3f''(a)^2 - 2f'''(a)f'(a)}{12f'(a)^2} \delta_n^3 + \dots$$

The number of digits of accuracy approximately triples with each iteration. We also require

$$3f''(a)^2 - 2f'''(a)f'(a) < 0.$$

Otherwise the iteration will be unstable.

Question break

Questions!

Make sure you understand how the above functions work, and the notion of stability.

Discuss in a group any questions you might have. Hopefully the group can explain it. If not, just ask!

To spark some ideas, here are some possible questions to consider:

- What's the best way to graphically show the basins of attraction? Can you implement it?
- What about showing convergence of a set of initial guesses?
- How fast does Newton's method converge for a double root?
- Why didn't the example from last weeks tutorial have second order convergence?
- Can you write a root-finding function using Newton's method? It should return a root given the following arguments
  - a function `f(x)`
  - its derivative `dfdx(x)`
  - an initial guess `x0`
  - an optional tolerance `eps`
- Can you adapt that algorithm to Halley's method?

You should try to answer these questions. Either now, or after the lecture.

IRREDUCIABLE COMPLEXITY IN MATHEMATICS*

The complexity found in this simple example can get philosophical real quick. But it's a useful kind of philosophy. This is a great summary of a lot of work on information and computation:

https://arxiv.org/abs/math/0411091 (https://arxiv.org/abs/math/0411091)

In this case the idea is simple: Is there a 'function' that encodes the basin of attraction for some value of a? Could we write this function down without just doing the 'computational experiment'? If yes, then we are very lucky. The more likely answer is that we just have to do it by hand.

There are many important and/or interesting and/or useful things in mathematics that can't be obtained except for actually doing the calculation.

*Note: this is ***not*** related to term that's used in biology.

What about Newton's method in $>$ 1D?

OK, we'll actually dig into this next time. But for now, let's look at the complex plane. It's kind of a "*poor man's 2D*".

It's a good idea to quit the kernel and start over in this example.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

Let's look for the roots of

$$f(z) = z^5 - 1$$

This means iterating the function

$$z_{n+1} = F(z_n), \quad \text{with} \quad F(z) = \frac{4\,z^5 + 1}{5\,z^4}$$

The initial guesses are now in the complex plane, $\mathbb{C}$.

In [2]:

```
def F(z):
    """Define iterating function F"""
    a=5
    return (1+(a-1)*(z**a))/(a*(z**(a-1)))

def iterate(f,z,k):
    """Function to recursively iterate any function f, k times"""
    if k == 0: return f(z)
    return iterate(f,f(z),k-1)

def iterate_proc(f,z,k):
    """Function to procedurally iterate any function f, k times"""
    ans = z
    if k > 0:
        for i in range(k):
            ans = f(ans)
    return ans

def which_root(domain,n=200,k=30,recursive=True):
    """Iterate F over section of complex plane."""
    x0, x1 = domain[0][0],domain[0][1]
    y0, y1 = domain[1][0],domain[1][1]

    x = np.linspace(x0,x1,n)
    y = np.linspace(y0,y1,n)

    x.shape = (len(x),1)
    y.shape = (1,len(y))

    z=x+1j*y

    if recursive: ans = iterate(F,z,k)
    else: ans = iterate_proc(F,z,k)

    return x+0*y,0*x+y,(5*np.log(ans)/(2*np.pi*1j)).real + 5
```

In [3]:

```
n = 1000
x0, x1, y0, y1 = -2, 2, -2, 2
x = np.linspace(x0,x1,n)
y = np.linspace(y0,y1,n)

x.shape = (len(x),1)
y.shape = (1,len(y))

z=x+1j*y
```

In [4]:

```
x,y,w = which_root([(-2,2),(-2,2)],n=400,recursive=True)
```

The white `x`'s show the location of the roots in the complex plane. The different colours correspond to different basins of attraction of the Newton iterator. Boundaries become very convoluted near the basin borders.

In [5]:

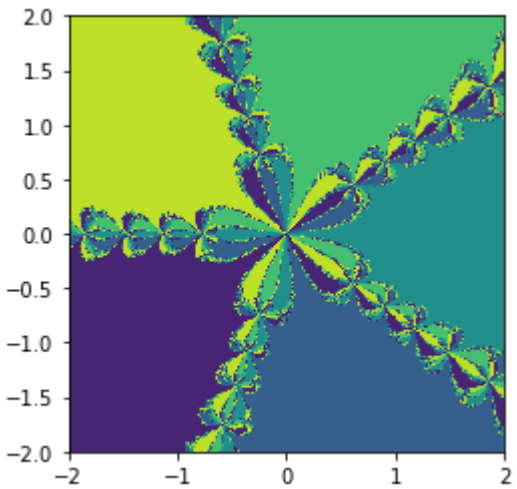
```
# %matplotlib notebook
%matplotlib inline
```

In [6]:

```
fig, ax = plt.subplots()
ax.set(aspect=1)
ax.pcolormesh(x[:,0],y[0,:],w.T)
# plt.savefig('complex.png',dpi=800,bbox_inches='tight')
```

Out[6]:

```
<matplotlib.collections.QuadMesh at 0x7f96a9a7c048>
```



In [7]:

```
def s(x):
    if np.sign(x)<0: return '-'
    return ' '
for n in range(5):
    x0  = np.cos(2*np.pi*n/5)
    y0  = np.sin(2*np.pi*n/5)
    ax.plot(x0,y0,marker="X",linestyle="",color="white")
    print('(x,y) = (%s%.2f,%s%.2f)' %(s(x0),abs(x0),s(y0),abs(y0)))

fig
```

```
(x,y) = ( 1.00, 0.00)
(x,y) = ( 0.31, 0.95)
(x,y) = (-0.81, 0.59)
(x,y) = (-0.81,-0.59)
(x,y) = ( 0.31,-0.95)
```

Out[7]: