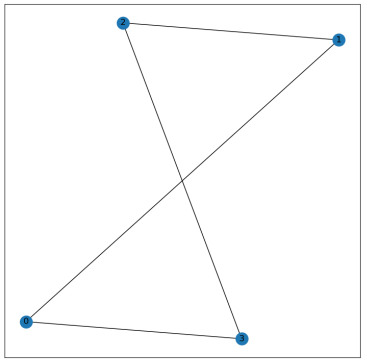
CT5141 Lab Week 11

James McDermott

Graph layout

1. (no solution needed)
2. A local optimum might look like this (suppose all edges want to be length 1):



As we can see, the desired lengths have not been achieved, but in order to move from here to a global optimum (just a square), we'd have to disimprove first.

3. A graph of 3 nodes that results in a local optimum: I don't think this can arise. However, a graph of 3 nodes that doesn't achieve its desired edge lengths: consider a "triangle" graph with desired edge lengths (1, 1, 4). This is an "impossible" triangle, so we will not achieve these lengths, even at the global optimum.
4. Consider each data point as a node in a fully connected graph, ie every node connected to every other. The desired edge-length between every pair of nodes is given by the distance in n-dimensional space. So, we create this graph and pass it in to our `graph_layout.py` code.

Gradient descent with various algorithms

We may see output like this, so let's interpret it:

```
      fun: 2.4868995751603507e-14
 hess_inv: array([[ 0.50126012, -0.49873988],
       [-0.49873988,  0.50126012]])
      jac: array([-2.38418579e-07, -2.38418579e-07])
  message: 'Optimization terminated successfully.'
     nfev: 30
      nit: 3
     njev: 10
   status: 0
  success: True
        x: array([-7.82960597e-09, -7.82960597e-09])
```

So, we see the optimization was successful. The best point (`x`) is close to $(0, 0)$, with an f value there very close to zero (`fun`). It took 3 iterations, but 30 f-evaluations. That's because we did not calculate the gradient in advance, instead `minimize` carried out *approximation* of the gradient, which means running f many extra times. The Jacobian (gradient) is a vector of length 2, close to zero (as expected at a minimum). In some second-order methods, we don't calculate the Hessian itself but we approximate the inverse of the Hessian (`hess_inv`).

Using the Jacobian

Above, `minimize` internally **approximates** the Jacobian (the gradient). But if we can calculate it exactly, we can use it:

With Sympy:

```
n = 2
A = 10
x = sympy.symarray("x", n)
rastrigin_sym = A*n + sum(xi**2 -
     A*sympy.cos(2*sympy.pi * xi) for xi in x)
rastrigin_df_dx0 = sympy.diff(rastrigin_sym, x[0])
```

The output `2*x_0 + 20*pi*sin(2*pi*x_0)` shows that the derivative wrt x_0 is $2x_0 + 20\pi \sin(2\pi x_0)$. Similarly, the derivative wrt x_1 is $2x_1 + 20\pi \sin(2\pi x_1)$. (You can check these on paper.)

Using the results above we can write:

```
def rastrigin_grad(x):
    # return a tuple (df/dx0, df/dx1)
    return 2*x[0] + 20*np.pi*np.sin(2*np.pi*x[0]), 2*x[1] + 20*np.pi*np.sin(2*np.pi*x[1])

scipy.optimize.minimize(rastrigin, x0, jac=rastrigin_grad)
```

Among other things we see that now `nfev = 10`. Now `minimize` doesn't have to approximate the Jacobian, but it still has to approximate the inverse hessian.

We could go a step further and calculate the Hessian (matrix of second partial derivatives) also, and pass them in, but it's a bit more complicated so we stop here.

If we want to do local optimization (no ability to avoid local optima) of real-valued functions, then `scipy.optimize.minimize` with or without passing in the Jacobian is a good first approach.