# Multidimensional-Newton

September 7, 2017

## 1 Newton's method and nonlinear equations

In first-year calculus, most students learn Newton's method for solving nonlinear equations $f(x) = 0$, which iteratively improves a sequence of guesses for the solution $x$ by **approximating f by a straight line**. That is, it **approximates a *nonlinear* equation by a sequence of approximate *linear* equations**.

This can be extended to *systems* of nonlinear equations as a **multidimensional Newton** method, in which we iterate by solving a sequence of linear (*matrix*) systems of equations. This is one example of an amazing fact: **linear algebra is a fundamental tool even for solving nonlinear equations**.

### 1.1 Packages for this notebook

```
In [1]: # Pkg.add.(["Interact", "PyPlot", "ForwardDiff"]) # uncomment this line to install packages
        using Interact, PyPlot
```

```
INFO: Recompiling stale cache file /Users/stevenj/.julia/lib/v0.5/PyPlot.ji for module PyPlot.
```

### 1.2 One-dimensional Newton

The standard one-dimensional Newton's method proceeds as follows. Suppose we are solving for a zero (root) of $f(x)$:

$$f(x) = 0$$

for an arbitrary (but differentiable) function $f$, and we have a guess $x$. We find an *improved* guess $x + \delta$ by Taylor expanding $f(x+\delta)$ around $x$ to *first order* (linear!) in $\delta$, and finding the . (This should be accurate if $x$ is *close enough* to a solution, so that the $\delta$ is *small*.) That is, we solve:

$$f(x + \delta) \approx f(x) + f'(x)\delta = 0$$

to obtain $\delta = -f(x)/f'(x)$. Plugging this into $x + \delta$, we obtain:

$$\boxed{\text{new } x = x - f(x)/f'(x)}.$$

This is called a **Newton step**. Then we simply repeat the process.

Let's visualize this process for finding a root of $f(x) = 2\cos(x) - x + x^2/10$ (a transcendental equation that has no closed-form solution).

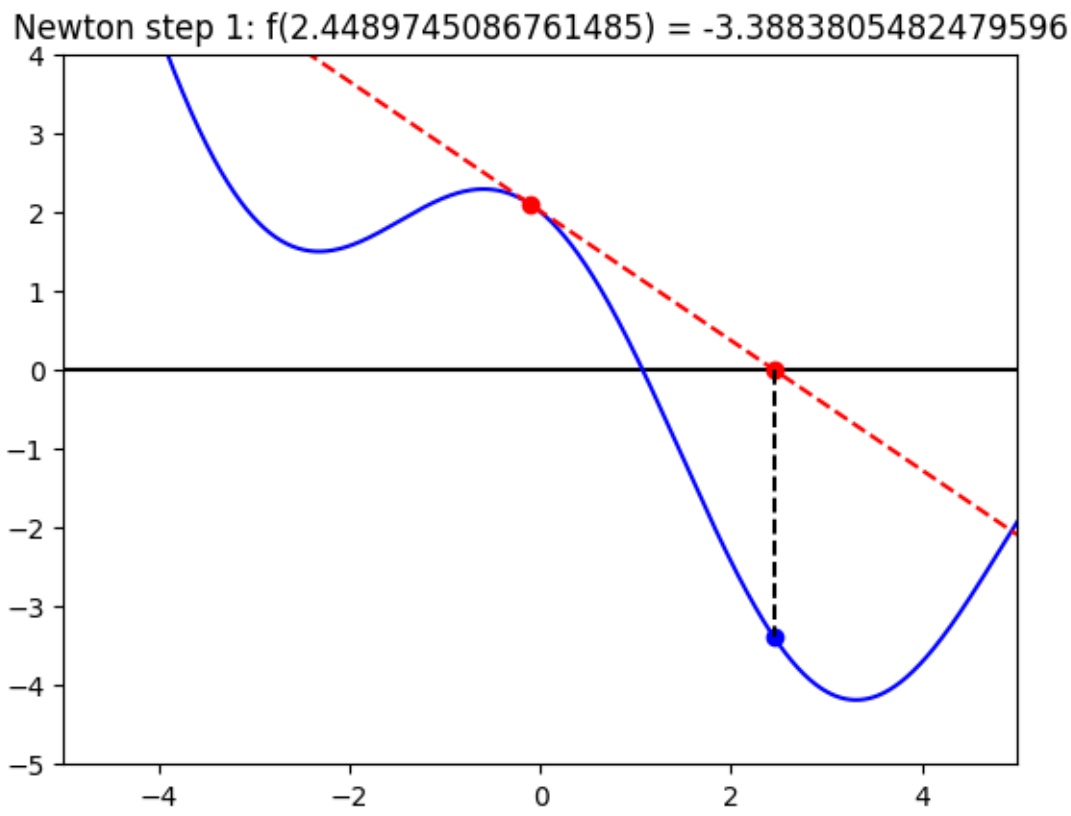
```
In [2]: fig = figure()
        xs = linspace(-5,5,1000)
        @manipulate for step in slider(1:20, value=1), start in slider(-4:0.1:4, value=-0.1)
            withfig(fig) do
                x = start
                local xprev, f, f′
                for i = 1:step
                    xprev = x
```

```
            f = 2cos(x) - x + x^2/10
            f′ = -2sin(x) - 1 + 2x/10
            x = x - f/f′
        end
        plot(xs, 0*xs, "k-")
        plot(xs, 2cos.(xs) - xs + xs.^2/10, "b-")
        newf = 2cos(x) - x + x^2/10
        plot([xprev, x], [f, 0.0], "ro")
        plot(x, newf, "bo")
        plot(xs, f + f′ * (xs - xprev), "r--")
        plot([x, x], [0, newf], "k--")
        xlim(minimum(xs), maximum(xs))
        ylim(-5,4)
        title("Newton step $step: f($x) = $newf")
    end
end
```

```
Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"",1,1:20,"horizontal",true,"d",true)
```

```
Interact.Slider{Float64}(Signal{Float64}(-0.1, nactions=1),"",-0.1,-4.0:0.1:4.0,"horizontal",true,".3f"
```

Out[2]:



If you start it anywhere near a root of $f(x)$, Newton's method can converge extremely quickly: **asymptotically, it doubles the number of accurate digits on each step**.

However, if you start it far from a root, the convergence can be hard to predict, and it may not even converge at all (it can oscillate forever around a local minimum).

Still, in many practical applications, there are ways to get a good initial guess for Newton, and then it is an extremely effective way to solve nonlinear equations to high accuracy.

## 1.3 A nonlinear circuit problem

Consider the nonlinear circuit graph from the graphs and networks lecture:

The incidence matrix $A$ of this graph is:

```
In [3]: A = [ -1   0   0   1   0   0
               0   0   0  -1   1   0
               0   0   0   0  -1   1
               0   0   1   0   0  -1
               0   1  -1   0   0   0
               1  -1   0   0   0   0
               0  -1   0   0   0   1
               0   0   0  -1   0   1 ]
```

```
Out[3]: 8×6 Array{Int64,2}:
         -1   0   0   1   0   0
          0   0   0  -1   1   0
          0   0   0   0  -1   1
          0   0   1   0   0  -1
          0   1  -1   0   0   0
          1  -1   0   0   0   0
          0  -1   0   0   0   1
          0   0   0  -1   0   1
```

### 1.3.1 Review: (Linear) circuit equations

Recall that if we associate a vector $v$ of voltages with the 6 nodes, then $d = Av$ gives the **voltage rise** across each edge, and $i = -YAv$ gives the **current** through each edge, where $Y$ is a diagonal matrix of admittances (= 1/resistance)

$$Y = \begin{pmatrix} Y_1 & & & \\ & Y_2 & & \\ & & \ddots & \\ & & & Y_8 \end{pmatrix}$$

This is simply an expression of Ohm's law.

Furthermore, we showed that Kirchhoff's current law is just the statement $A^T i = 0$. Putting it all together, and including a current source term $s$ (an external current flowing out of each node), we obtained the equations:

$$A^T Y A v = s$$

where to have a solution ($A^T Y A$ is singular) we had to have $s \perp N(A)$, or equivalently $\sum_i s_i = 0$: the net current flowing in/out of the circuit must be zero.

### 1.3.2 Nonlinear Ohm's law

A key physical foundation of the equations above was Ohm's law: $i_j = -d_j/R_j = -Y_j d_j$, which is the statement that the **current is proportional to the voltage drop** $-d_j$.

However, this is only an approximation. In reality, as the voltage and current increase, the resistance changes. For example, the resistor heats up (and eventually melts!) due to the dissipated power $i_j^2/Y_j = Y_j d_j^2$, and resistance depends on temperature.

Let's try a **simple model** of a voltage-dependent resistance. Suppose that we modify Ohm's law to:

$$i_j = -\underbrace{\frac{Y_j}{1+\alpha_j d_j^2}}_{\tilde{Y}_j(d_j)} d_j$$

where $\tilde{Y}_j(d_j) = Y_j/(1+\alpha_j d_j^2)$ corresponds to a resistance that increases quadratically with the voltage rise $d_j$. This model is not unrealistic! For a real resistor, you could measure the voltage dependence of $Y$, and *fit* it to an equation of this form, and it would be valid for sufficiently small $d_j$! (The admittance will normally only depend on $d^2$, not on $d$, because with most materials it will not depend on the sign of the voltage drop or current.)

The problem, of course, is that with this nonlinear Ohm's law, the whole problem becomes a **nonlinear system of equations**. How can we solve such a system of equations? At first glance, the methods of 18.06 don't work — they are only for *linear* systems.

**Newton's method: Linearizing the equation** The trick is the same as Newton's method. We suppose that we have a *guess* $v$ for the voltages, and hence a guess $d = Av$ for the voltage drops. Now, we want to find an *improved guess* $v+\delta$, and we find $\delta$ by *linearizing* the equations in $\delta$: just a multidimensional Taylor expansion.

That is, we are trying to find a root of:

$$f(v) = A^T \tilde{Y}(Av) Av - s$$

where $\tilde{Y}(Av)$ is the diagonal matrix of our nonlinear admittances from above. We Taylor expand this to first order:

$$f(v+\delta) \approx f(v) + J(v)\delta$$

where $J(v)$ is some $n \times n$ matrix (a Jacobian matrix, in fact). Then we solve for our step $\delta$:

$$\delta = -J(v)^{-1} f(v)$$

and finally we get:

$$\text{new } v = v + \delta = v - J(v)^{-1} f(v)$$

Clearly, $J(v)$ is the analogue of $f'(x)$ in the one-dimensional Newton method. We will look at the general formula for $J(v)$ below, but to start with let's work it out in this particular case. Of course, in practice we won't actually invert $J$: we'll solve $J(v)y = f(v)$ by `J\f` (elimination/LU) or some other method.

**The Jacobian matrix for nonlinear admittance** The admittance is written in terms of the voltage rise $d = Av$. If we change $v$ to $v+\delta$, then we get $d + A\delta$. Let's denote $\hat{\delta} = A\delta$. Then the formula for each component of our current $i$ becomes:

$$i_j = -\tilde{Y}_j(d_j + \hat{\delta}_j)\,(d_j + \hat{\delta}_j) \approx -\tilde{Y}_j(d_j) d_j - (\tilde{Y}_j(d_j) + \tilde{Y}_j'(d_j) d_j)\hat{\delta}_j$$

where we have just Taylor-expanded around $\hat{\delta}_j = 0$, and $\tilde{Y}_j'(d_j) = -2\alpha_j d_j Y_j/(1+\alpha_j d_j^2)^2$. Let $K_j(d_j) = \tilde{Y}_j(d_j) + \tilde{Y}_j'(d_j) d_j$, and then we have:

$$i_j \approx -\left[\tilde{Y}_j(d_j) d_j + K_j(d_j)\hat{\delta}_j\right]$$

Plugging this into $f(v+\delta)$, we get

$$f(v+\delta) \approx \underbrace{A^T Y(Av)Av - s}_{f(v)} + \underbrace{A^T K(Av)A}_{J(v)}\,\delta$$

and hence

$$J(v) = A^T K(Av) A$$

where

$$K(d) = \begin{pmatrix} K_1(d_1) & & & \\ & K_2(d_2) & & \\ & & \ddots & \\ & & & K_8(d_8) \end{pmatrix}$$

There is one slight problem here: $J$ is a singular matrix even if all $K_j \neq 0$, because of $N(A) \neq \{0\}$. Just as with $s$, $J(v)y = f(v)$ only has a solution if $f(v) \in C(J) \subseteq C(A^T)$. Fortunately, $f(v) = A^T Y A - s$ is *always* in $C(A^T)$ as long as $s \in C(A^T) = N(A)^\perp$, which was required *anyway* (from above) if we are to have a solution in the *linear* case.

(We can still run into a singular $J$ for "unlucky" $K_j$, but that is a typical hazard of Newton's method, just like we can run into $f'(x) = 0$ for unlucky $x$ values. The important thing is that it doesn't happen "generically", i.e. singularities only occur at isolated points.)

**Example** For an example, let's just set $Y_k = 1$ and $\alpha_k = 0.5$ for $k = 1, \ldots, 6$, and use $s = (1, -1, 0, 0, 0, 0)$ as in the previous lecture (current injected into node 2 and extracted out from node 1).

What should we use as our initial guess? How about the solution to the *linear* problem with $\tilde{Y}(0) = Y$? That's often a good guess, since in many real problems the nonlinear solution will be very close to the solution of a simplified linear problem.

However, in this case, let's use an even simpler first guess: $v = 0$. Then our first Newton step will be new $v = -J(0)^{-1} f(0) = (A^T Y A)^{-1} s$, since $f(0) = -s$ and $K(0) = Y$. Of course, by the "matrix inverse × vector" here we really mean "solve this system of equations with that right hand side" — this is especially important here because the inverse doesn't even exist (the matrix is singular), but a solution exists (the right-hand-side is in the column space).

Let's write some code to compute $f(v)$ and $J(v)$:

```
In [4]: Y[U+2096] = 1
        α[U+2096] = 0.5
        Y~[U+2096](d) = Y[U+2096] / (1 + α[U+2096] * d^2)
        Y~[U+2096]′(d) = -2α[U+2096]*d*Y[U+2096] / (1 + α[U+2096] * d^2)^2
        K[U+2096](d) = Y~[U+2096](d) + Y~[U+2096]′(d)*d

        f(v) = A' * (diagm(Y~[U+2096].(A*v)) * (A*v)) - [1,-1,0,0,0,0]
        J(v) = A' * diagm(K[U+2096].(A*v)) * A

Out[4]: J (generic function with 1 method)
```

Now let's implement the Newton step $v - J(v)^{-1} f(v)$.

I'd ideally like to use `v - J(v) \ f(v)` in Julia, but the \ function will complain that `J` is singular, even though `f` is in the column space. There are various ways to do this properly numerically, but for simplicity I will "cheat" and use and advanced tool called the pseudo-inverse, computed in julia by `pinv(J)`, which (for a right-hand-side in the column space) will give us a particular solution similar to what we would get from the elimination technique we learned in class. (We will cover the pseudo-inverse much later in 18.06.) There are other ways to proceed here that are even more efficient, but `pinv` is the simplest.

```
In [5]: newtonstep(v) = v - pinv(J(v)) * f(v)
```

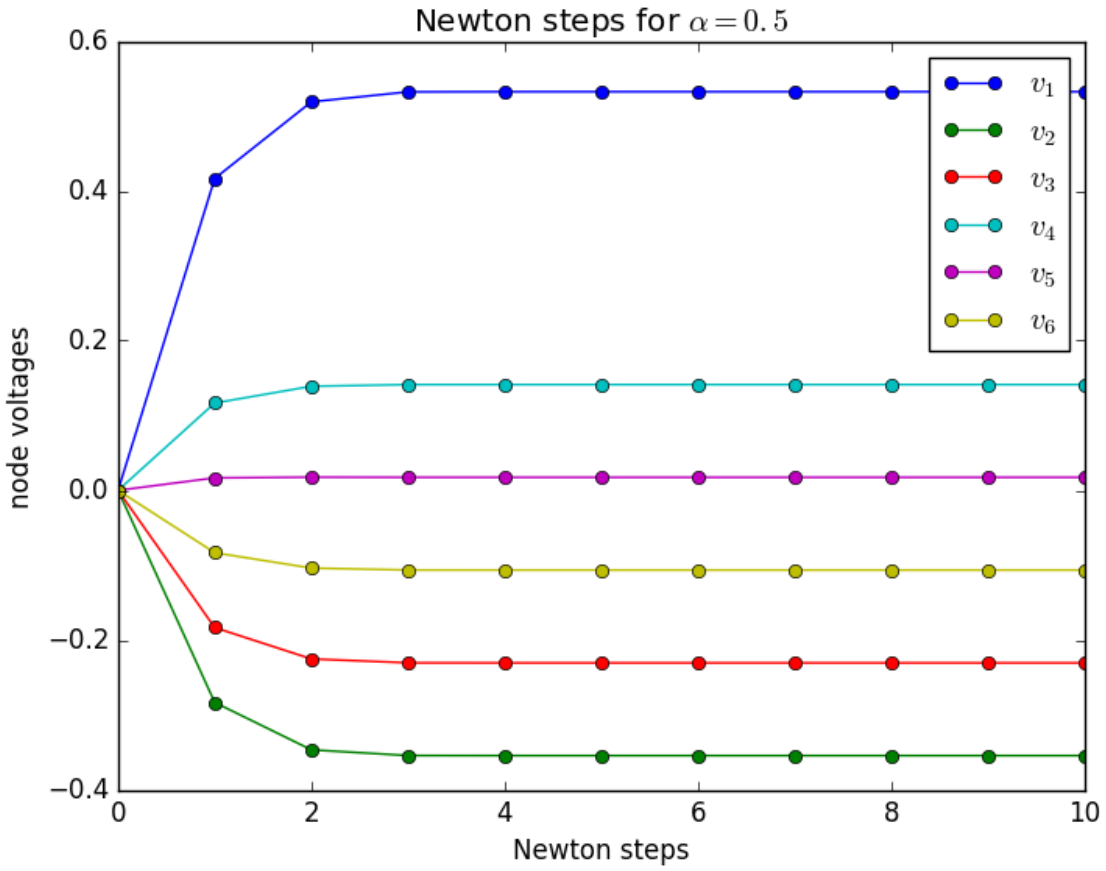
```
Out[5]: newtonstep (generic function with 1 method)
```

Now, let's see what happens. We'll start by just plotting the voltages as a function of the Newton step, to see how (and whether) it is converging:

```
In [6]: function newton(v, nsteps)
            for i = 1:nsteps
                v = newtonstep(v)
            end
            return v
        end
        newton(v, nsteps::AbstractVector) = map(n -> newton(v,n), nsteps)
```

```
Out[6]: newton (generic function with 2 methods)
```

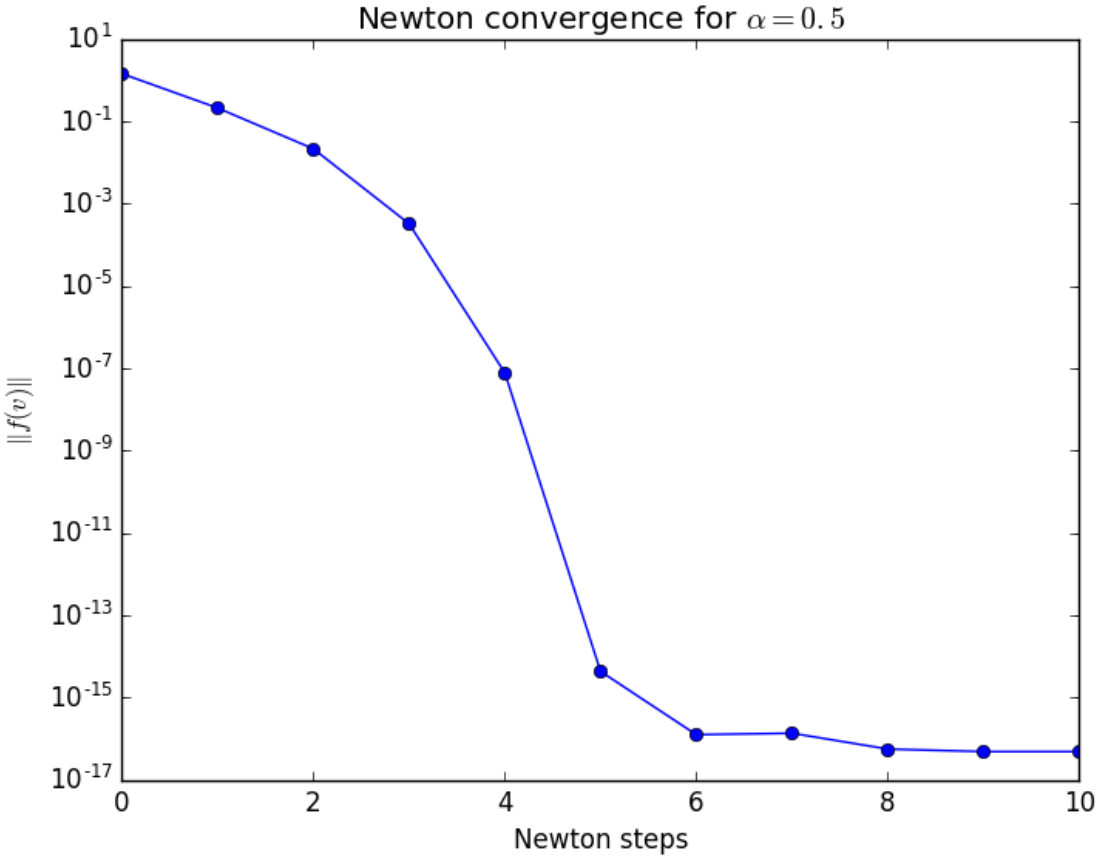
```
In [7]: vsteps = newton(zeros(6), 0:10)
        plot([vsteps[i][j] for i=1:length(vsteps), j=1:6], "o-")
        xlabel("Newton steps")
        ylabel("node voltages")
        legend([L"v_1", L"v_2", L"v_3", L"v_4", L"v_5", L"v_6"], loc="upper right")
        title(L"Newton steps for $\alpha = 0.5$")
```



```
Out[7]: PyObject <matplotlib.text.Text object at 0x323bc39d0>
```

Clearly, it is converging pretty rapidly. Another way to see this is to plot the convergence of the length (norm) of the $f(v)$ vector, $\|f(v)\| = \sqrt{f(v)^T f(v)}$:

```
In [8]: xlabel("Newton steps")
        ylabel(L"\Vert f(v) \Vert")
        semilogy(norm.(f.(vsteps)), "bo-")
        title(L"Newton convergence for $\alpha = 0.5$")
```




```
Out[8]: PyObject <matplotlib.text.Text object at 0x323f6d310>
```

It converges **faster than exponentially** with the step. Once it is close to the solution, Newton roughly **doubles the number of digits in each step**.

Eventually, it stops getting better: the accuracy is **limited by roundoff errors** once the error reaches $\approx 10^{-16}$, related to the fact that the computer is only doing arithmetic to about 16 digits of accuracy.

## 1.4 General Multidimensional Newton

The general case of the multidimensional Newton algorithm is as follows. We are solving:

$$\begin{pmatrix} f_1(x_1, \ldots, x_n) \\ f_2(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix} = f(x) = 0$$

for $x \in \mathbb{R}^n$ and $f(x) \in \mathbb{R}^n$: $n$ (possibly nonlinear but differentiable) equations in $n$ unknowns.

Given a guess $x$, we want to find an improved guess $x + \delta$ for $\delta \in \mathbb{R}^n$. We do this by Taylor-expanding $f$ around $x$ to first order (linear):

$$f(x + \delta) \approx f(x) + J(x)\,\delta = 0$$

where $J$ is the $n \times n$ Jacobian matrix with entries $\boxed{J_{ij} = \partial f_i / \partial x_j}$, i.e.

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

Hence, we solve the *linear* equation

$$J(x)\,\delta = -f(x)$$

for the Newton step $\delta$, obtaining (if $J$ is invertible):

$$\boxed{x + \delta = x - J(x)^{-1} f(x)}$$

Some things to remember:

- Newton converts $n$ *nonlinear* equations into repeated solution of $n \times n$ *linear* equations.
- When solving nonlinear equations, coming up with a good initial guess is a bit of an art, that often requires some problem-specific understanding. A typical trick is to solve a related problem, e.g. a linear problem. Or to start with a linear problem and to "turn on" the nonlinearity gradually, using the solution of each nonlinear problem as the starting guess for the next one.
- If you start too far from a root, Newton's method can sometimes take a large step, far outside the validity of the Taylor approximation, and this can actually make the guess *worse*. Sophisticated implementations use a variety of techniques to make the convergence more robust, such as a backtracking line search or a trust region. These techniques are outside the scope of 18.06, though!
- There are other methods related to Newton that don't require you to compute $J(x)$ at all. You only supply $f(x)$ and they either approximate the Jacobian for you directly (e.g. Broyden's method) or implicitly (e.g. Anderson acceleration). There is a rich mathematical literature on solution methods for nonlinear systems of equations, but essentially all the methods have one thing in common: *linear* algebra plays a key role.

## 1.5 Automatic differentiation

The Jacobian matrix can often be *automatically* computed from $f(x)$ by the computer using automatic differentiation tools, saving you from the tedious (and error-prone) task of writing out $J(x)$ manually.

In Julia, there are packages ForwardDiff and ReverseDiff to do this for you.

For example, let's compute the Jacobian of our $f(v)$ function above, and compare it to the $J(v)$ that we defined manually:

```
In [9]: using ForwardDiff

In [10]: v = [0.1,1.2,3.4,5.6,-0.3,0.7] # a "random" vector
         ForwardDiff.jacobian(f, v)

Out[10]: 6×6 Array{Float64,2}:
           0.0990134  -0.153337   0.0        0.0543237   0.0         0.0
          -0.153337    0.72329    0.121405   0.0         0.0        -0.691358
           0.0         0.121405  -0.243995   0.0         0.0         0.12259
```

```
 0.0543237   0.0         0.0        -0.167821    0.0484289   0.0650683
 0.0         0.0         0.0         0.0484289   0.173793   -0.222222
 0.0        -0.691358    0.12259     0.0650683  -0.222222    0.725922
```

```
In [11]: ForwardDiff.jacobian(f, v) - J(v)
```

```
Out[11]: 6×6 Array{Float64,2}:
  5.55112e-17  -5.55112e-17   0.0          ...   0.0           0.0
 -5.55112e-17   0.0           0.0               0.0           0.0
  0.0           0.0          -5.55112e-17       0.0           5.55112e-17
  0.0           0.0           0.0              -2.77556e-17   2.77556e-17
  0.0           0.0           0.0               2.77556e-17   0.0
  0.0           0.0           5.55112e-17  ...   0.0          -1.11022e-16
```

Yup, it computed our Jacobian automatically and perfectly accurately (up to the machine-precision rounding errors).

How ForwardDiff works, based on dual numbers, is outside the scope of 18.06, but there is a nice tutorial notebook by David Sanders.