Week 9: Monday, Mar 28

Problem du jour

Suppose $r : \mathbb{R} \to \mathbb{R}^n$. What is the formula for Newton iteration on

$$\phi(x) = \frac{1}{2}\|r(x)\|^2 = \frac{1}{2}r(x)^T r(x)?$$

Answer: We need first and second derivatives:

$$\begin{aligned}\phi'(x) &= r'(x)^T r(x) \\ \phi''(x) &= r'(x)^T r'(x) + r''(x)^T r(x)\end{aligned}$$

So Newton iteration is

$$x_{k+1} = x_k - \left[r'(x_k)^T r'(x_k) + r(x_k)^T r''(x_k)\right]^{-1} \left[r'(x_k)^T r(x_k)\right].$$

Notice that if $r(x_*) \approx 0$ then near x_* we have

$$\phi''(x) \approx r'(x)^T r'(x).$$

If we use this approximation, we get *Gauss-Newton iteration*

$$\begin{aligned}x_{k+1} &= x_k - \left[r'(x_k)^T r'(x_k)\right]^{-1} \left[r'(x_k)^T r(x_k)\right] \\ &= x_k - r'(x_k)^\dagger r(x_k).\end{aligned}$$

Where ordinary Newton iteration on a system of nonlinear equations solves a linear system at each step, Gauss-Newton iteration for a nonlinear least squares problem solves a linear least squares problem at each step. You will get the chance to solve a problem with Gauss-Newton iteration in HW 5.

Logistics

- HW 4 and project 2 are graded (see CMS)
- HW 5 is due via CMS at 11:59 next Monday
- Up next: interpolation, differentiation, and integration
- Prelim 2 is 4/7: least squares; nonlinear equations and optimization; interpolation, numerical differentiation, and integration

Function approximation

A common task in scientific computing is to approximate a function. The approximated function might be available only through tabulated data, or it may be the output of some other numerical procedure, or it may be the solution to a differential equation. The approximating function is usually chosen because it is relatively simpler to evaluate and analyze. Depending on the context, we might want an approximation that is accurate for a narrow range of arguments (like a Taylor series), or we might want guaranteed global accuracy over a wide range of arguments. We might want an approximation that preserves properties like monotonicity or positivity (e.g. when approximating a probability density). We might want to exactly match measurements at specified points, or we might want an approximation that "smooths out" noisy data. We might care a great deal about the cost of forming the approximating function if it is only used a few times, or we might care more about the cost of evaluating the approximation after it has been formed. There are a huge number of possible tradeoffs, and it is worth keeping these types of questions in mind in practice.

Though function approximation is a huge subject, we will mostly focus on approximation by polynomials and piecewise polynomials. In particular, we will concentrate on *interpolation*, or finding (piecewise) polynomial approximating functions that exactly match a given function at specified points.

Polynomial interpolation

This is the basic polynomial interpolation problem: given data $\{(x_i, y_i)\}_{i=0}^d$ where all the t_i are distinct, find a degree d polynomial $p(x)$ such that $p(x_i) = y_i$ for each i. Such a polynomial always exists and is unique.

The Vandermonde approach

Maybe the most obvious way to approach to this problem is to write

$$p(x) = \sum_{j=0}^{d} c_j x^j,$$

where the unknown x_j are determined by the interpolation conditions

$$p(x_i) = \sum_{j=0}^{d} c_j x_i^j = y_j.$$

In matrix form, we can write the interpolation conditions as

$$Ac = y$$

where $a_{ij} = x_i^j$ (and we're now thinking of the index j as going from zero to d). The matrix A is a *Vandermonde matrix*. The Vandermonde matrix is nonsingular, and we can solve Vandermonde systems using ordinary Gaussian elimination in $O(d^3)$ time.

You may vaguely recall having seen this sort of Vandermonde system in HW 2, problem 2, early in the semester. If you go back and read the footnote in the solution set, though, you will see that I commented that this is usually a bad way to compute things numerically. The problem is that the condition numbers of Vandermonde systems grow exponentially with the system size, yielding terribly ill-conditioned problems even for relatively small problems.

The Lagrange approach

The problem with the Vandermonde matrix is not in the basic setup, but in how we chose to represent the space of degree d polynomials. In general, we can write

$$p(x) = \sum_{j=0}^{d} c_j q_j(x)$$

where $\{q_j(x)\}$ is some other basis for the space of polynomials of degree at most d. The power basis $\{x^j\}$ just happens to be a poor choice from the perspective of conditioning.

One alternative to the power basis is a basis of *Lagrange polynomials*:

$$L_i(x) = \frac{\prod_{j \neq i}(x - x_i)}{\prod_{j \neq i}(x_j - x_i)}.$$

The polynomial L_i is characterized by the property

$$L_i(x_j) = \begin{cases} 1, & j = i \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, if we write the interpolating polynomial in the form

$$p(x) = \sum_{j=0}^{d} c_j L_j(x),$$

the interpolation conditions yield the linear system

$$Ic = y,$$

i.e. we simply have

$$p(x) = \sum_{j=0}^{d} y_j L_j(x),$$

It is trivial to find the coefficients in a representation of an interpolant via Lagrange polynomials. But what if we want to evaluate the Lagrange form of the interpolant at some point? The most obvious algorithm costs $O(d^2)$ per evaluation, which is more expensive than the $O(d)$ cost of evaluating a polynomial in the usual monomial basis using Horner's rule.

Horner's rule

There are typically two tasks in applications of polynomial interpolation. The first task is getting some representation of the polynomial; the second task is to actually evaluate the polynomial. In the case of the power basis $\{x^j\}_{j=0}^d$, we would usually evaluate the polynomial in $O(d)$ time using Horner's method. You have likely seen this method before, but it is perhaps worth going through it one more time.

Horner's scheme can be written in terms of a recurrence, writing $p(x)$ as $p_0(x)$ where

$$p_j(x) = c_j + xp_{j+1}(x)$$

and $p_d(x) = c_d$. For example, if we had three data points, we would write

$$\begin{aligned}
p_2(x) &= c_2 \\
p_1(x) &= c_1 + xp_2(x) = c_1 + xc_2 \\
p_0(x) &= c_0 + xp_1(x) = c_0 + xc_1 + x^2c_2.
\end{aligned}$$

Usually, we would just write a loop:

```
function px = peval(c,x)
  px = c(end)*x;
  for j = length(c)-1:-1:1
    px = c(j) + x.*px;
  end
```

But even if we would usually write the loop with no particular thought to the recurrence, it is worth remembering how to write the recurrence.

Yhe idea of Horner's rule extends to other bases. For example, suppose we now write a quadratic as

$$p(x) = c_0 q_0(x) + c_1 q_1(x) + c_2 q_2(x).$$

An alternate way to write this is

$$p(x) = q_0(c_0 + q_1/q_0(c_1 + c_2 q_2/q_1));$$

more generally, we could write $p(x) = q_0(x)p_0(x)$ where $p_d(x) = c_d$ and

$$p_j(x) = c_j + p_{j+1}(x)q_{j+1}(x)/q_j(x).$$

In the case of the monomial basis, this is just Horner's rule, but the recurrence holds more generally.

The Newton approach

The Vandermonde approach to interpolation requires that we solve an ill-conditioned linear system (at a cost of $O(d^3)$) to find the interpolating polynomial. It then costs $O(d)$ per point to evaluate the polynomial. The Lagrange approach gives us a trivial linear system for the coefficients, but it then costs $O(d^2)$ per point to evaluate the resulting representation. Newton's form of the interpolant will give us a better balance: $O(d^2)$ time to find the coefficients, $O(d)$ time to evaluate the function.

Newton's interpolation scheme uses the polynomial basis

$$\begin{aligned} q_0(x) &= 1 \\ q_j(x) &= \prod_{k=1}^{j} (x - x_k), \quad j > 0. \end{aligned}$$

If we write

$$p(x) = \sum_{j=0}^{d} c_j q_j(x),$$

the interpolating conditions have the form

$$Uc = y,$$

where U is an upper triangular matrix with entries

$$u_{ij} = q_j(x_i) = \prod_{k=j}^{d} (t_i - t_j)$$

for $i = 0, \ldots d$ and $j = 0, \ldots, d$. Because U is upper triangular, we can compute the coefficients c_j in $O(d^2)$ time; and we can use the relationship $q_j(x) = (x - x_j)q_{j-1}(x)$ as the basis for a Horner-like scheme to evaluate $p(x)$ in $O(d)$ time (this is part of a problem on HW 5).

In practice, we typically do not form the matrix U in order to compute x. Instead, we express the components of x in terms of *divided differences.* That is, we write

$$c_j = y[x_1, \ldots, x_{j+1}]$$

where the coefficients $y[x_i, \ldots, x_j]$ are defined recursively by the relationship

$$\begin{aligned} y[x_i] &= y_i, \\ y[x_i, x_{i+1}, \ldots, x_j] &= \frac{y[x_i, x_{i+1}, \ldots, x_{j-1}] - y[x_{i+1}, \ldots, x_j]}{x_i - x_j}. \end{aligned}$$

Evaluating the x_j coefficients by divided differences turns out to be numerically preferable to forming U and solving by back-substitution.