MAT321 Numerical Methods

Department of Mathematics and PACM
Princeton University

Instructor: Nicolas Boumal (nboumal)
TAs: Thomas Pumir (tpumir), Eitan Levin (eitanl)

Fall 2019

Contents

Introduction

> "Numerical analysis is the study of algorithms for the problems of continuous mathematics."
>
> –Nick Trefethen, appendix of [TBI97]

These notes contain some of the material covered in MAT 321 / APC 321 – Numerical Methods taught at Princeton University during the Fall semesters of 2016–2019. They are extensively based on the two reference books of the course, namely,

- Endre Süli and David F. Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003, and
- Lloyd N. Trefethen and David Bau III. *Numerical linear algebra*, volume 50. SIAM, 1997.

I thank Bart Vandereycken and Javier Gómez-Serrano, previous instructors of this course, and Pierre-Antoine Absil for their help and insight. Special thanks also to José S.B. Ferreira, who was my TA for the first two years, to Yuan Liu who took on that role in 2018, and to Thomas Pumir and Eitan Levin for 2019.

These notes are work in progress (this is their third year.) Please do let me know about errors, typos, suggestions for improvements... (however small.) Your feedback is immensely welcome, always.

Nicolas Boumal

Chapter 1

Solving one nonlinear equation

The following problem is arguably ubiquitous in science and engineering:

Problem 1.1 (Nonlinear equation)**.** *Given* $f\colon [a,b] \to \mathbb{R}$*, continuous, find* $\xi \in [a,b]$ *such that* $f(\xi) = 0$*.*

The assumption that f is continuous is of central importance. Indeed, without that assumption, evaluating f at $x \in [a,b]$ yields no information whatsoever about the value of f at any other point in the interval: unless $f(x) = 0$, we are not in a better position to solve the problem. Compare this to the situation where f is continuous: then, if we query f at $x \in [a,b]$ we know at least that *close* to x, the value of f must be *close* to $f(x)$. In particular, if $|f(x)|$ is small, there might be a root nearby. This is enough to get started. Later, we will assume a stronger form of continuity, called *Lipschitz continuity*: this will quantify what we mean by "close".

Consider the following example: $f(x) = e^x - 2x - 1$, depicted in Figure 1.1. To get a sense of what is possible, let's take a look at how Matlab's built-in algorithm, `fzero`, behaves, when given the hint to search close to $x_0 = 1$:

```
f = @(x) exp(x) - 2*x - 1;
x0 = 1;
options = optimset('Display','iter');
xi = fzero(f, x0, options);
fprintf('Root found: xi = %.16e, with value f(xi) = ...
    %.6e.\n', xi, f(xi));
```

This produces the following output:

```
Search for an interval around 1 containing a sign change:
Func-count    a          f(a)             b          f(b)        Procedure
 1                1     -0.281718             1     -0.281718   initial interval
 3         0.971716     -0.300957       1.02828     -0.260304   search
 5             0.96     -0.308304          1.04     -0.250783   search
 7         0.943431     -0.318082       1.05657     -0.236654   search
 9             0.92      -0.33071          1.08      -0.21532   search
11         0.886863     -0.346223       1.11314     -0.182382   search
13             0.84     -0.363633          1.16     -0.130067   search
15         0.773726     -0.379623       1.22627     -0.044042   search
17             0.68     -0.386122          1.32      0.103421   search

Search for a zero in the interval [0.68, 1.32]:
Func-count    x          f(x)             Procedure
17             1.32      0.103421        initial
18          1.18479     -0.099576        interpolation
19          1.25112   -0.00799173        interpolation
20          1.25649   8.62309e-05        interpolation
21          1.25643  -5.34422e-07        interpolation
22          1.25643  -3.53615e-11        interpolation
23          1.25643             0        interpolation

Zero found in the interval [0.68, 1.32]
Root found: xi = 1.2564312086261697e+00, with value f(xi) = 0.000000e+00.
```

Based on our initial guess $x_0 = 1$, Matlab's `fzero` used 23 function evaluations to zoom in on the positive root of f.

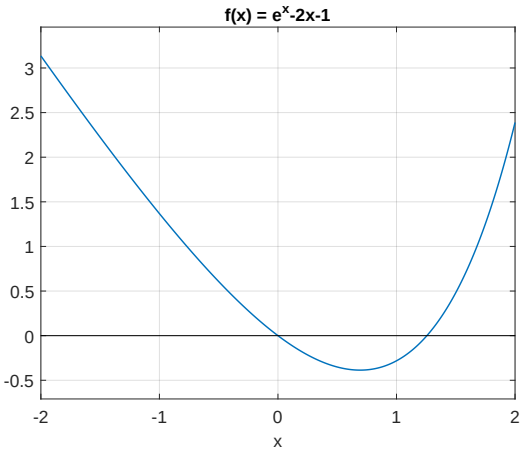
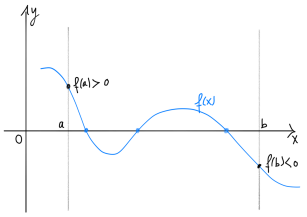


Figure 1.1: The function $f(x) = e^x - 2x - 1$ has two roots: $\xi = 0$ and $\xi \approx 1.2564312086261697$.

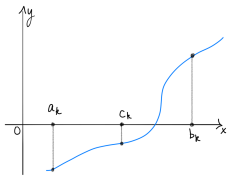
1.1 Bisection

The main theorem we need to describe our first algorithm is a consequence of the Intermediate Value Theorem (IVT). It offers a sufficient (but not necessary) criterion to decide whether Problem 1.1 has a solution at all.

Theorem 1.2. *Let $f\colon [a,b] \to \mathbb{R}$ be continuous. If $f(a)f(b) \leq 0$, then there exists $\xi \in [a,b]$ such that $f(\xi) = 0$.*



Proof. If $f(a)f(b) = 0$, then either a or b can be taken as ξ. Otherwise, $f(a)f(b) < 0$, so that $f(a)$ and $f(b)$ delimit an interval which contains 0. Apply the IVT to conclude. □



Hence, if we find two points in the interval $[a,b]$ such that f changes sign on those two points, we are assured that f has a root in between these two points. Without loss of generality, say that a, b are two such points (alternatively, we can always redefine the domain of f.) Say that $f(a) < 0$ and $f(b) > 0$. Let's evaluate f at the midpoint $c = \frac{a+b}{2}$. What could happen?

- $f(c) = 0$: then we return $\xi = c$;
- $f(c) > 0$: then f changes sign on $[a,c]$;
- $f(c) < 0$: then f changes sign on $[c,b]$.

In both last cases, we identified an interval which (i) contains a root, and (ii) is twice as small as our original interval. By iterating this procedure, we can repeatedly halve the length of our interval with a single function evaluation, always with the certainty that this interval contains a solution to our problem. After k iterations, the interval has length $|b-a|2^{-k}$. The midpoint of that interval is at a distance at most $|b-a|2^{-k-1}$ of a solution ξ. We formalize this in Algorithm 1.1, called the *bisection algorithm.*

Theorem 1.3. *When Algorithm 1.1 returns c, there exists $\xi \in [a_0, b_0]$ such that $f(\xi) = 0$ and $|\xi - c| \leq |b_0 - a_0|2^{-1-K}$. Assuming $f(a_0), f(b_0)$ were already computed, this is achieved in at most K function evaluations.*

In principle, if we iterate the bisection algorithm indefinitely, we should reach an arbitrarily accurate approximation of a root ξ of f. While this statement is mathematically correct, it does not square with practice: see Figures 1.2 and 1.3. Indeed, by default, computers use a form of inexact arithmetic known as the IEEE Standard for Floating-Point Arithmetic (IEEE 754)—more on that later.

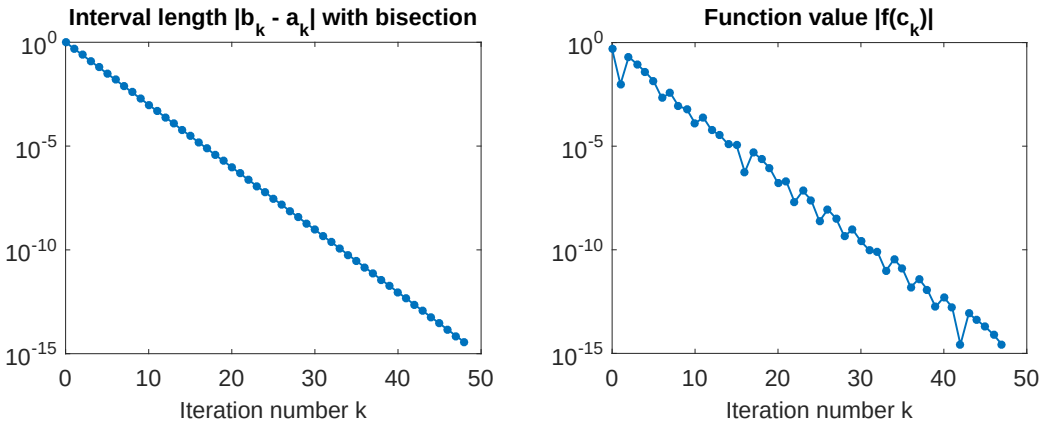


Figure 1.2: Applying the bisection algorithm on $f(x) = e^x - 2x - 1$ with $[a_0, b_0] = [1, 2]$ and $K = 60$. The interval length $\ell_k = b_k - a_k$ decreases by a factor of 2 at each iteration, and the function value eventually hits 0 with $c_{48} = 1.2564312086261697$ (zero is not represented on the log-scale). Yet, computing $f(c_{48})$ with high accuracy shows it is not quite a root. Using Matlab's symbolic computation toolbox, `syms x; f = exp(x)- ... 2*x - 1; vpa(subs(f, x, 1.2564312086261697), 20)` gives $f(c_{48}) \approx 1.086 \cdot 10^{-16}$: a small error remains. Figure 1.3 show a different scenario.

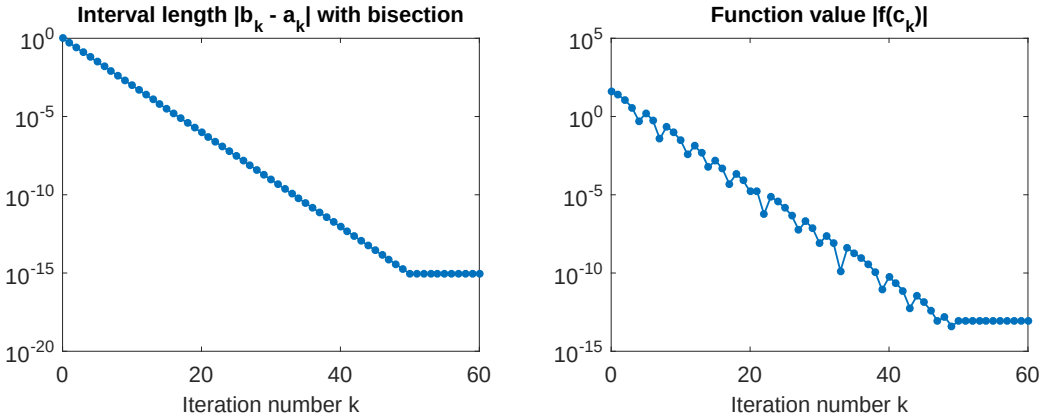


Figure 1.3: Bisection on $f(x) = (5-x)e^x - 5$ with $[a_0, b_0] = [4, 5]$ and $K = 60$. The interval length $\ell_k = b_k - a_k$ decreases only to $8.9 \cdot 10^{-16}$. Furthermore, the function value stagnates instead of converging to 0. The culprit: inexact arithmetic. We will see that this is actually as accurate as one can hope.

Algorithm 1.1 Bisection

Input: $f\colon [a_0, b_0] \to \mathbb{R}$, continuous, $f(a_0)f(b_0) < 0$; iteration budget K
Let $c_0 = \frac{a_0+b_0}{2}$
for $k = 0, 1, 2 \ldots, K-1$ **do**
    Compute $f(c_k)$ ▷ We really only need the sign
    if $f(c_k)$ has sign opposite to $f(a_k)$ **then**
        Let $(a_{k+1}, b_{k+1}) = (a_k, c_k)$
    else if $f(c_k)$ has sign opposite to $f(b_k)$ **then**
        Let $(a_{k+1}, b_{k+1}) = (c_k, b_k)$
    else
        return $c = c_k$ ▷ $f(c_k) = 0$
    end if
    Let $c_{k+1} = \frac{a_{k+1}+b_{k+1}}{2}$
end for
return $c = c_K$ ▷ We ran out of iteration budget

```
function c = my_bisection(f, a, b, K)
% Example: c = my_bisection(@(x) exp(x) - 2*x - 1, 1, 2, 60);

% Make sure a and b are distinct and a < b
assert(a ~= b, 'a and b must be different');
if b < a
    [a, b] = deal(b, a); % switch a and b
end

% Two calls to f here
fa = f(a);
fb = f(b);

% Return immediately if a or b is a root
if fa == 0
    c = a;
    return;
end
if fb == 0
    c = b;
    return;
end

assert(sign(fa) ~= sign(fb), 'f(a) and f(b) must have ...
   opposite signs');

c = (a+b)/2;
```

```
for k = 1 : K

    % Only one call to f per iteration
    fc = f(c);

    if fc == 0
        return;              % f(c) = 0: done
    end

    if sign(fc) ~= sign(fa)  % update interval to [a, c]
        b = c;
        fb = fc;
    else                     % update interval to [c, b]
        a = c;
        fa = fc;
    end

    c = (a+b)/2;

end
```

The bisection algorithm relies heavily on Theorem 1.2. For its many qualities (not the least of which is its simplicity), this approach has three main drawbacks:

1. The user needs to find a sign change interval $[a_0, b_0]$ as initialization;
2. Convergence is fast, but we can do better;
3. Theorem 1.2 is fundamentally a one-dimensional thing: it won't generalize when we aim to solve several nonlinear equations simultaneously.

In the next section, we discuss *simple iterations*: a family of iterative algorithms designed to solve Problem 1.1, and which will (try to) address these shortcomings.

Recall the performance of `fzero` reported at the beginning of this chapter. Based on our initial guess $x_0 = 1$, Matlab's `fzero` used 17 function evaluations to find a sign-change interval of length 0.64. After that, it needed only 6 additional function evaluations to find the same root our bisection found in 48 iterations (starting from that interval, bisection would reach an error bound of 0.005: only two digits after the decimal point are correct.) If we give `fzero` the same interval we gave bisection, then it needs only 10 function evaluations to do its job. This confirms Problem 1.1 can be solved faster. We won't discuss how `fzero` finds a sign-change interval too much (you will think about it during precept). We do note in Figure 1.4 that this can be a difficult task. The methods we discuss next do not require a sign-change interval.

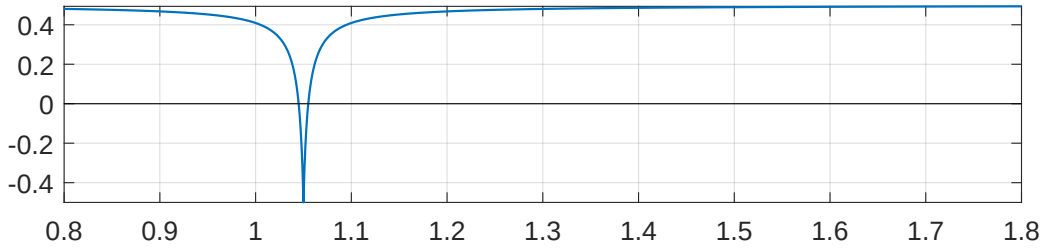


Figure 1.4: Plot of `f = @(x).5 - 1./(1 + M*abs(x - 1.05));` with `M = 200;`. Given an intial guess $x_0 = 1$, Matlab's `fzero` aims to find a sign-change interval: after 4119 function evaluations, it abandons, with the last considered interval being $[-1.6 \cdot 10^{308}, 1.6 \cdot 10^{308}]$. On the other hand, Matlab's `fsolve` finds an excellent approximation of the root in 18 function evaluations, from the same intialization: run `x0 = 1; options = optimset('Display','iter'); ... fzero(f, x0, options); fsolve(f, x0, options);`

1.2 Simple iteration

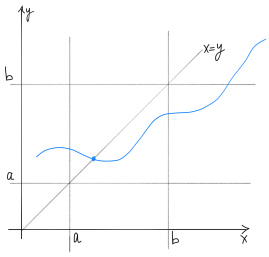
The family of algorithms we describe now relies on a different criterion for the existence of a solution to Problem 1.1. As an example, consider

$$g(x) = x - f(x).$$

Clearly, $f(\xi) = 0$ if and only if $g(\xi) = \xi$, that is, if ξ is a *fixed point* of g. Given f, there are many ways to construct a function g whose fixed points coincide with the roots of f, so that Problem 1.1 is equivalent to the following.

Problem 1.4. *Given* $g\colon [a,b] \to \mathbb{R}$ *continuous, find* $\xi \in [a,b]$ *s.t.* $g(\xi) = \xi$.

Brouwer's theorem states a sufficient condition for the existence of a fixed point. (Note the condition on the image of g.)



Theorem 1.5 (Brouwer's fixed point theorem)**.** *If* $g\colon [a,b] \to [a,b]$ *is continuous, then there exists (at least one)* $\xi \in [a,b]$ *such that* $g(\xi) = \xi$.

Proof. We can reduce the statement to that of Theorem 1.2 by defining $f(x) = x - g(x)$. Indeed, $f(a) = a - g(a) \leq 0$ since $g(x) \geq a$ for all x. Likewise, $f(b) \geq 0$. Thus, $f(a)f(b) \leq 0$ and Theorem 1.2 allows to conclude. □

If one exists, finding a fixed point of g *can* be rather easy, see Algorithm 1.2. Given an initial guess $x_0 \in [a,b]$, this algorithm generates a

Algorithm 1.2 Simple iteration

1: **Input:** $g\colon [a,b] \to [a,b]$, continuous; initial guess $x_0 \in [a,b]$.
2: **for** $k = 0, 1, 2 \ldots$ **do**
3: $\quad x_{k+1} = g(x_k)$
4: **end for**

sequence $x_0, x_1, x_2 \ldots$ in $[a,b]$ by iterated application of g: $x_{k+1} = g(x_k)$. (Notice here the importance that g maps $[a,b]$ to itself, so that it is always possible to apply g to the new iterate.) By continuity, we get an easy statement right away.

Theorem 1.6. *If the sequence $x_0, x_1, \ldots$ produced by Algorithm 1.2 converges to a point ξ, then $g(\xi) = \xi$.*

Proof. $\xi = \lim_{k\to\infty} x_k = \lim_{k\to\infty} x_{k+1} = \lim_{k\to\infty} g(x_k) \overset{\text{continuity}}{=} g\left(\lim_{k\to\infty} x_k\right) = g(\xi).$ □

This theorem has a big "if". The main concern for this section will be: Given f as in Problem 1.1, how do we pick an appropriate function g so that (i) simple iteration on g converges, and (ii) it converges fast.

Let's do an example, with $f(x) = e^x - 2x - 1$, as in Figure 1.1. Here are three possible functions g_i which all satisfy $f(\xi) = 0 \iff g_i(\xi) = \xi$:

$$\begin{aligned} g_1(x) &= \log(2x+1), \\ g_2(x) &= \frac{e^x - 1}{2}, \\ g_3(x) &= e^x - x - 1. \end{aligned} \tag{1.1}$$

(The domain of g_1 is restricted to $(-1/2, \infty)$.) See Figure 1.5. Notice how $g_1([1,2]) \subset [1,2]$ and $g_{2,3}([-1/2, 1/2]) \subset [-1/2, 1/2]$: fixed points exist.

Let's run simple iteration with these functions and see what happens. First, initialize all three sequences with $x_0 = 0.5$ and run 20 iterations.

```
x = zeros(20, 3);                    % run 20 iterations for each
x(1, :) = 0.5;                       % initialize

for k = 1 : size(x, 1)-1
    x(k+1, 1) = g1(x(k, 1));
    x(k+1, 2) = g2(x(k, 2));
    x(k+1, 3) = g3(x(k, 3));
end

fprintf('      g1              g2              g3\n');
fprintf('%10.8e\t%10.8e\t%10.8e\n', x');
```

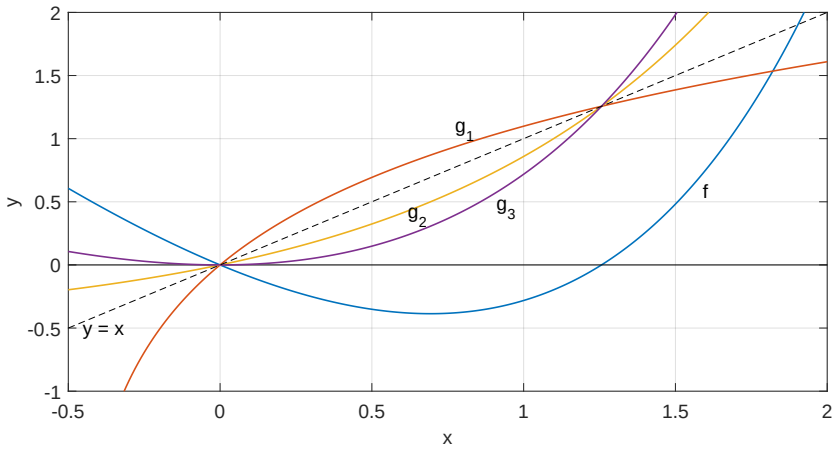


Figure 1.5: Functions g_i intersect the line $y = x$ (that is, $g_i(x) = x$) exactly when $f(x) = 0$.

This produces the following output.

```
      g1              g2              g3
5.00000000e-01 5.00000000e-01 5.00000000e-01
6.93147181e-01 3.24360635e-01 1.48721271e-01
8.69741686e-01 1.91573014e-01 1.16282500e-02
1.00776935e+00 1.05576631e-01 6.78709175e-05
1.10377849e+00 5.56756324e-02 2.30328290e-09
1.16550958e+00 2.86273445e-02 0.00000000e+00
1.20327831e+00 1.45205226e-02 0.00000000e+00
1.22570199e+00 7.31322876e-03 0.00000000e+00
1.23878110e+00 3.67001786e-03 0.00000000e+00
1.24633153e+00 1.83838031e-03 0.00000000e+00
1.25066450e+00 9.20035584e-04 0.00000000e+00
1.25314261e+00 4.60229473e-04 0.00000000e+00
1.25455714e+00 2.30167698e-04 0.00000000e+00
1.25536366e+00 1.15097094e-04 0.00000000e+00
1.25582323e+00 5.75518590e-05 0.00000000e+00
1.25608500e+00 2.87767576e-05 0.00000000e+00
1.25623408e+00 1.43885858e-05 0.00000000e+00
1.25631897e+00 7.19434467e-06 0.00000000e+00
1.25636731e+00 3.59718527e-06 0.00000000e+00
1.25639483e+00 1.79859587e-06 0.00000000e+00
```

Recall that the larger root is about 1.2564312086261697. Let's try again with

initialization $x_0 = 1.5$.

```
      g1              g2              g3
1.50000000e+00 1.50000000e+00 1.50000000e+00
1.38629436e+00 1.74084454e+00 1.98168907e+00
1.32776143e+00 2.35107853e+00 4.27329775e+00
1.29623914e+00 4.74844242e+00 6.64845880e+01
1.27884229e+00 5.72021964e+01 7.47979509e+28
1.26910993e+00 3.47991180e+24        Inf
1.26362374e+00        Inf         NaN
1.26051782e+00        Inf         NaN
1.25875516e+00        Inf         NaN
1.25775344e+00        Inf         NaN
1.25718372e+00        Inf         NaN
1.25685955e+00        Inf         NaN
1.25667505e+00        Inf         NaN
1.25657003e+00        Inf         NaN
1.25651024e+00        Inf         NaN
1.25647620e+00        Inf         NaN
1.25645683e+00        Inf         NaN
1.25644579e+00        Inf         NaN
1.25643951e+00        Inf         NaN
1.25643594e+00        Inf         NaN
```

Question 1.7. *Explain why the g_3 sequence generates NaN's (Not-a-Number) after the first Inf (∞).*

■

Now $x_0 = 10$.

```
      g1              g2              g3
1.00000000e+01 1.00000000e+01 1.00000000e+01
3.04452244e+00 1.10127329e+04 2.20154658e+04
1.95855062e+00        Inf         Inf
1.59271918e+00        Inf         NaN
1.43161144e+00        Inf         NaN
1.35150178e+00        Inf         NaN
1.30914426e+00        Inf         NaN
1.28600113e+00        Inf         NaN
1.27312630e+00        Inf         NaN
1.26589144e+00        Inf         NaN
```

```
1.26180281e+00        Inf        NaN
1.25948479e+00        Inf        NaN
1.25816821e+00        Inf        NaN
1.25741966e+00        Inf        NaN
1.25699381e+00        Inf        NaN
1.25675147e+00        Inf        NaN
1.25661353e+00        Inf        NaN
1.25653500e+00        Inf        NaN
1.25649030e+00        Inf        NaN
1.25646485e+00        Inf        NaN
```

The sequence generated by g_1 converges reliably to the larger root, slowly. The sequence by g_2, if it converges, converges to 0, also slowly. The sequence by g_3, when it converges, converges to 0 very fast. Much of these differences can be explained with the concept of *contractions* and the associated theorem.

Definition 1.8 (contraction)**.** *Let $g\colon [a,b] \to \mathbb{R}$ be continuous. We say g is a contraction if there exists $L \in (0,1)$ such that*

$$\forall x, y \in [a,b], \quad |g(x) - g(y)| \leq L|x - y|.$$

In words: g brings x, y closer; this is a type of Lipschitz condition.

If g maps $[a,b]$ to itself *and* it is a contraction, it is easy to establish convergence of simple iteration. The role of L and the importance of having $L \in (0,1)$ become apparent in the proof.

Theorem 1.9 (contraction mapping theorem)**.** *Let $g\colon [a,b] \to [a,b]$ be continuous. If g is a contraction, then it has a unique fixed point ξ and the simple iteration sequence $x_0, x_1 \ldots$ generated by $x_{k+1} = g(x_k)$ converges to ξ for any $x_0 \in [a,b]$.*

Proof. The proof is in three steps.

1. A fixed point ξ exists, by Theorem 1.5.

2. The fixed point is unique. By contradiction: if $\xi' = g(\xi')$ and $\xi' \neq \xi$, then

$$|\xi - \xi'| = |g(\xi) - g(\xi')| \overset{\text{Def. 1.8}}{\leq} L|\xi - \xi'|.$$

   Since $\xi' \neq \xi$, we get $L \geq 1$, which is a contradiction for a contraction.

3. Convergence: $|x_{k+1} - \xi| = |g(x_k) - g(\xi)| \leq L|x_k - \xi|$. By induction, it follows that $|x_k - \xi| \leq L^k|x_0 - \xi|$. Since $L \in (0, 1)$, this converges to zero as k goes to infinity, hence $\lim_{k\to\infty} x_k = \xi$. □

From the last step of the proof, we also get a sense that having L closer to zero should translate into faster convergence. Let's investigate whether functions g_i from our example are contractions, and if so, with which constants L_i. First, recall the Mean Value Theorem (MVT).

Theorem 1.10 (MVT)**.** *If $g\colon [a, b] \to \mathbb{R}$ is continuous and it is differentiable on (a, b), then there exists $\eta \in (a, b)$ such that $g(b) - g(a) = g'(\eta)(b - a)$.*

Consider the MVT and the definition of contraction. If $g\colon [a, b] \to [a, b]$ is continuous and it is differentiable on (a, b), then for all $x, y \in [a, b]$, we have $|g(x) - g(y)| = |g'(\eta)||x - y|$ for some $\eta \in (a, b)$. Thus, replacing $|g'(\eta)|$ with a bound independent of η (and independent of x and y), we reach the conclusion that

$$\forall x, y \in [a, b], \quad |g(x) - g(y)| \leq L|x - y|,$$

with

$$L \triangleq \sup_{\eta\in(a,b)} |g'(\eta)|.$$

If this quantity is in $(0, 1)$, then g is a contraction on $[a, b]$. (Note that the sup over (a, b) is equivalent to a max over $[a, b]$ if g' is continuous on $[a, b]$.)

Are the functions g_i contractions? Yes, on some intervals. Consider Figure 1.6 which depicts $|g_i'(x)|$. We have:

- $g_1([1, 2]) \subset [1, 2]$ and $L_1 = \max_{\eta\in[1,2]} |g_1'(\eta)| \leq 0.667$.
- $g_2([-1/2, 1/2]) \subset [-1/2, 1/2]$ and $L_2 = \max_{\eta\in[-1/2,1/2]} |g_2'(\eta)| \leq 0.825$.
- $g_3([-1/2, 1/2]) \subset [-1/2, 1/2]$ and $L_3 = \max_{\eta\in[-1/2,1/2]} |g_3'(\eta)| \leq 0.649$.

Theorem 1.9 (the contraction mapping theorem) guarantees convergence to a unique fixed point for these g_i's, given appropriate initialization. What can be said about the speed of convergence? Consider the proof of Theorem 1.9. In the last step, we established $|x_k - \xi| \leq L^k|x_0 - \xi|$. What does it take to ensure $|x_k - \xi| \leq \varepsilon$? Certainly, if $L^k|x_0 - \xi| \leq \varepsilon$, we are in the clear. Taking logarithms, this is the case if and only if:

$$\begin{aligned} k\log(L) + \log|x_0 - \xi| &\leq \log(\varepsilon) && \text{(multiply by } -1\text{)} \\ k\log(1/L) &\geq \log(|x_0 - \xi|) + \log(1/\varepsilon) \\ k &\geq \frac{1}{\log(1/L)} \log\left(\frac{|x_0 - \xi|}{\varepsilon}\right). \end{aligned}$$

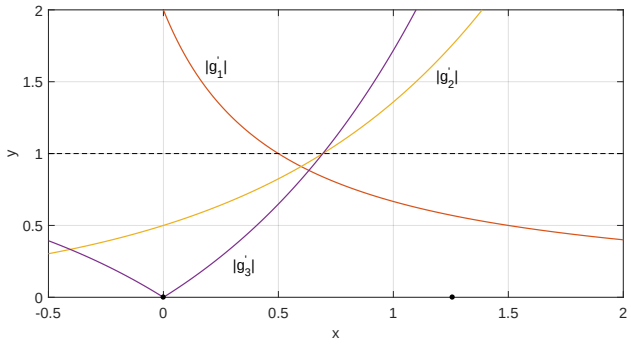


Figure 1.6: Absolute values of the derivatives of functions g_i. For a differentiable function g_i to be a contraction around x, a necessary condition is that $|g_i'(x)| < 1$. Black dots mark the roots of f.

Of course, we do not know ξ. Surely, $|x_0 - \xi| \leq [a, b]$, but in practice we also rarely know $[a, b]$. Luckily, we can get around that by studying the first iterate:[1]

$$\begin{aligned} |x_0 - \xi| &\leq |x_0 - x_1| + |x_1 - \xi| \\ &= |x_0 - x_1| + |g(x_0) - g(\xi)| \\ &\leq |x_0 - x_1| + L|x_0 - \xi|. \end{aligned}$$

Thus, $|x_0 - \xi| \leq \frac{1}{1-L}|x_0 - x_1|$: assuming we know L, this is a computable quantity. Combining, we get the following bound on k.

Theorem 1.11. *Under the assumptions and with the notations of the contraction mapping theorem, with $x_0 \in [a, b]$, for all $k \geq k(\varepsilon)$ where*

$$k(\varepsilon) = \frac{1}{\log(1/L)} \log\left(\frac{|x_0 - x_1|}{(1-L)\varepsilon}\right),$$

it holds that $|x_k - \xi| \leq \varepsilon$.

This last theorem only gives an upper bound on how many iterations might be necessary to reach a desired accuracy. In practice, convergence may be much faster. Take for example g_3, which converged to the root 0 (exactly) in only 5 iterations when initialized with $x_0 = 0.5$. Meanwhile, the bound with $L_3 = 0.649$ only guarantees an accuracy of $L_3^5|x_0 - \xi| \approx 0.058$ for 5 iterations. Why is that?

[1] In the first line, we use the triangle inequality: https://en.wikipedia.org/wiki/Triangle_inequality#Example_norms.

One important reason is that the constant L is valid for a whole interval $[a, b]$. Yet, this choice of interval is somewhat arbitrary. If $x_k \to \xi$, eventually, it is really only g' close to ξ which matters. For g_1, the derivative g_1' evaluated at the positive root is about 0.57: not a big difference from 0.667. But for g_3, we have $g_3'(0) = 0$—as we get closer and closer to 0, the convergence gets faster and faster!

Thus, informally, if g is continuously differentiable at ξ and $x_k \to \xi$, asymptotically, the rate depends on $g'(\xi)$. In fact, much of the behavior of simple iteration is linked to $g'(\xi)$. Consider the following definition.

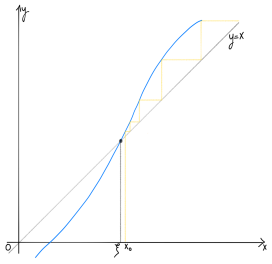
Definition 1.12. *Let $g\colon [a, b] \to [a, b]$ have a fixed point ξ, and let $x_0, x_1 \ldots$ be a sequence generated by $x_{k+1} = g(x_k)$ for some $x_0 \in [a, b]$.*

- *If there exists a neighborhood I of ξ such that $x_0 \in I$ implies $x_k \to \xi$, we say ξ is a* stable fixed point.
- *If there exists a neighborhood I of ξ such that $x_0 \in I \backslash \{\xi\}$ implies we do* not *have $x_k \to \xi$, we say ξ is an* unstable fixed point.

Consider $f(x) = \begin{cases} \frac{1}{2}x & \text{if } x \le 0, \\ 2x & \text{otherwise.} \end{cases}$

(ξ can be either or neither.)

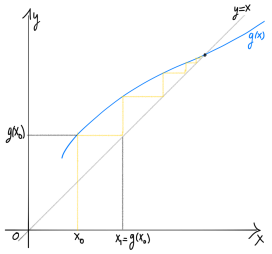
For a *continuously differentiable* function g with fixed point ξ, we can make the following statements (note that their "if" parts are quite different in nature.)



- If $|g'(\xi)| > 1$, then ξ is unstable. Indeed, if x_k is very close to ξ (but not equal!), then, by the MVT,

$$|x_{k+1} - \xi| = |g(x_k) - g(\xi)| = |g'(\eta)||x_k - \xi|$$

for some η between x_k and ξ. By continuity of g', we have $|g'(\eta)| > 1$ for η sufficiently close to ξ, hence: we are being pushed away from ξ by the iteration.



- If $x_k \to \xi$, then, by the MVT and by continuity of $|g'(x)|$,

$$\begin{aligned} \lim_{k\to\infty} \frac{|x_{k+1} - \xi|}{|x_k - \xi|} &= \lim_{k\to\infty} \frac{|g(x_k) - g(\xi)|}{|x_k - \xi|} = \lim_{k\to\infty} \frac{|g'(\eta_k)||x_k - \xi|}{|x_k - \xi|} \\ &= \lim_{k\to\infty} |g'(\eta_k)| = \left|g'\left(\lim_{k\to\infty} \eta_k\right)\right| = |g'(\xi)|, \end{aligned}$$

where η_k lies between x_k and ξ.

This last statement shows explicitly how $|g'(\xi)|$ drives the error decrease, asymptotically. If $g'(\xi) = 0$, convergence is mighty fast, eventually. Let's give some names to the convergence speeds we may encounter.

Definition 1.13. *Assume $x_k \to \xi$. We say:*

- *x_k converges to ξ* at least linearly *if there exists $\mu \in (0,1)$ and there exists $\varepsilon_0, \varepsilon_1 \ldots > 0$ such that $\varepsilon_k \to 0$, $|x_k - \xi| \leq \varepsilon_k$ and $\lim_{k\to\infty} \frac{\varepsilon_{k+1}}{\varepsilon_k} = \mu$.*
- *If the conditions hold with $\mu = 0$, the convergence is* superlinear.
- *If they hold with $\mu = 1$ and $|x_k - \xi| = \varepsilon_k$, the convergence is* sublinear.

For the first case, if furthermore $|x_k - \xi| = \varepsilon_k$, we say convergence is *linear*, and $\rho = -\log_{10}(\mu)$ is the *asymptotic rate of convergence*. This is because the number of correct digits of x_k as an approximation of ξ grows as $k\rho$ asymptotically (think about it), hence the term linear convergence.

It is a good idea to go back to Figures 1.5 and 1.6 to reinterpret the experiments in light of our understanding of the role of $|g'(\xi)|$.

At this point, a word of caution is necessary: these notions of convergence rates are *asymptotic*. When does the asymptotic regime kick in? That is largely unspecified. Consider

$$\begin{aligned} g_1(x) &= 0.99x, \\ g_2(x) &= \frac{x}{(1 + x^{1/10})^{10}}. \end{aligned}$$

Running a simple iteration on both functions from $x_0 = 1$ generates the following sequences. For g_1, we have linear convergence to 0:

$$x_k = 0.99^k, \text{ with } \rho = -\log_{10}(0.99) \approx 0.004,$$

and for g_2 we have sublinear convergence to 0:

$$x_k = \frac{1}{(k+1)^{10}}, \text{ and } \lim_{k\to\infty} \frac{|x_{k+1} - 0|}{|x_k - 0|} = \lim_{k\to\infty} \left(\frac{k+1}{k+2}\right)^{10} = 1.$$

Thus, convergence to 0 is eventually faster with g_1, yet as Figure 1.7 shows, this asymptotic behavior only kicks in after many thousands of iterations. For practical applications, the early convergence to a "good enough" approximation of the solution may be all that matters. (This is especially true for optimization algorithms in machine learning applied to very large datasets.)

1.3 Relaxation and Newton's method

Given a function f, we saw that different choices of g such that $g(x) = x \iff f(x) = 0$ lead to different behavior. Is there a systematic approach to pick g? Here is one.

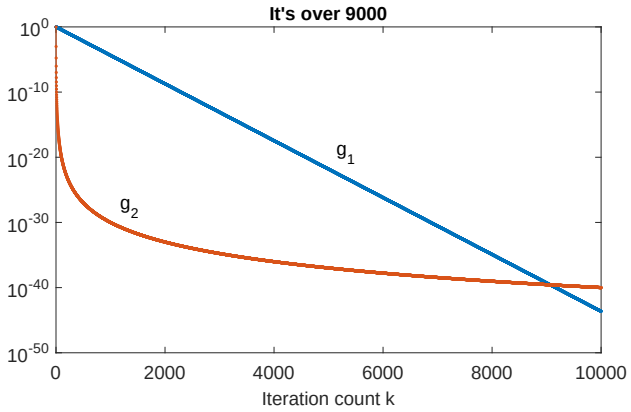


Figure 1.7: Even though the sequence generated by g_2 converges only sublinearly, it takes over 9000 iterations for the linearly convergent sequence generated by g_1 to take over.

Definition 1.14. *Let f be defined and continuous around ξ.* Relaxation *defines the sequence*

$$x_{k+1} = x_k - \lambda f(x_k),$$

where $\lambda \neq 0$ is to be chosen (see below) and x_0 is given near ξ.

Thus, relaxation is simple iteration with $g(x) = x - \lambda f(x)$. Since g is continuous, if relaxation converges to ξ, then $f(\xi) = 0$.

What about rates of convergence? Assuming differentiability, ideally, we want $|g'(\xi)| = |1 - \lambda f'(\xi)| < 1$. This is the case if and only if $1 - \lambda f'(\xi) < 1$ and $1 - \lambda f'(\xi) > -1$, that is:

- $f'(\xi) \neq 0$: the root is *simple*;
- λ and $f'(\xi)$ have the same sign; and
- $|\lambda|$ is not too big: using $\lambda f'(\xi) = |\lambda||f'(\xi)|$, we need $|\lambda| < \frac{2}{|f'(\xi)|}$.

Thus, if ξ is a simple root of f and f is continuously differentiable around ξ, there exists some $\lambda \neq 0$ such that relaxation converges at least linearly to ξ if started close enough: this statement is formalized in [SM03, Thm. 1.7].

The above reduces the construction of g to picking λ. Let's automate this as well. If we are optimistic, we may try to pick λ such that $g'(\xi) = 1 - \lambda f'(\xi) = 0$, hence set $\lambda = \frac{1}{f'(\xi)}$. The issue is that we do not know $f'(\xi)$. One idea that turns out to be particularly powerful is to allow λ to change with k. At every iteration, our best guess for ξ is x_k. So, let's use that and define $\lambda_k = \frac{1}{f'(x_k)}$.

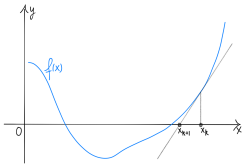
Definition 1.15. *For a given* x_0, Newton's method *generates the sequence:*

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

We implicitly assume that $f'(x_k) \neq 0$ *for all* k.

Question 1.16. *Show that, at every step,* x_{k+1} *is the root of the first-order Taylor approximation of* f *around* x_k.

■



If Newton's method converges (that's a big if!), then the rate is superlinear provided $f'(\xi) \neq 0$. Indeed, Newton's method is simple iteration with:

$$g(x) = x - \frac{f(x)}{f'(x)}, \qquad g'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2}.$$

Thus, $g'(\xi) = 0$. How fast exactly is this superlinear convergence? Let's look at an example on $f(x) = e^x - 2x - 1$:

```
f  = @(x) exp(x) - 2*x - 1;
df = @(x) exp(x) - 2;          % f' is easy to get here

% Initialization x_0: play around with this value: can get ...
   convergence to either root!
x = .5;
fprintf('x = %+.16e, \t f(x) = %+.16e\n', x, f(x));

for k = 1 : 12
    x = x - f(x) / df(x);
    fprintf('x = %+.16e, \t f(x) = %+.16e\n', x, f(x));
end
```

This produces the following output:

```
x = +5.0000000000000000e-01,   f(x) = -3.5127872929987181e-01
x = -5.0000000000000000e-01,   f(x) = +6.0653065971263342e-01
x = -6.4733401606416163e-02,   f(x) = +6.6784120574507444e-02
x = -1.8885640405095216e-03,   f(x) = +1.8903462554580308e-03
x = -1.7777391536067094e-06,   f(x) = +1.7777407337327134e-06
x = -1.5802248762500354e-12,   f(x) = +1.5802914532514478e-12
x = +6.6576998915534905e-17,   f(x) = -1.1102230246251565e-16
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
```

```
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
x = -4.4445303546980749e-17,   f(x) = +0.0000000000000000e+00
```

We get *fast* convergence to the root $\xi = 0$. After a couple iterations, the error $|x_k - \xi|$ appears to be squared at every iteration, until we run into an error of 10^{-17}, which is an issue of numerical accuracy. (As a practical concern, it is nice to observe that, if we keep iterating, we do not move away from this excellent approximation of the root.) Let's give a name to this kind of fast convergence.

Definition 1.17. *Suppose $x_k \to \xi$. We say the sequence $x_0, x_1 \ldots$ converges to ξ* with at least order $q > 1$ *if there exists $\mu > 0$ and a sequence $\varepsilon_0, \varepsilon_1 \ldots > 0$ with $\varepsilon_k \to 0$ such that*

$$|x_k - \xi| \leq \varepsilon_k \qquad \text{and} \qquad \lim_{k\to\infty} \frac{\varepsilon_{k+1}}{\varepsilon_k^q} = \mu.$$

If the inequality holds with equality, we say convergence is with order q*; if this holds with $q = 2$, the convergence is* quadratic.

A couple remarks are in order:

1. There is no need to require $\mu < 1$ since $q > 1$ (think about it.)

2. It makes no sense to discuss the rate of convergence of a sequence to ξ if it does not converge to ξ, which is why the definition above requires that the sequence converges to ξ as an assumption. Indeed, consider the following sequence: $x_k = 2$ for all k, and consider the limit $\lim_{k\to\infty} \frac{|x_{k+1}-0|}{|x_k-0|^2} = \frac{1}{2} > 0$. Of course, we *cannot* conclude from this that $x_0, x_1, x_2 \ldots$ converges to 0 quadratically, since it does not even converge to 0 in the first place. In fewer words: always secure convergence to ξ before your discuss the rate of convergence to ξ.

Question 1.18. *Show that simple iteration with g_3, if it converges to $\xi = 0$, does so quadratically.*

■

Finding such a g_3 was rather lucky. Newton's method, on the other hand, provides a systematic way of getting quadratic convergence to isolated roots (when it provides convergence), making it one of the most important algorithms in numerical analysis.

Theorem 1.19. *Let $f\colon \mathbb{R} \to \mathbb{R}$ be continuous with $f(\xi) = 0$. Assume $f''(x)$ is continuous in $I_\delta = [\xi - \delta, \xi + \delta]$ for some $\delta > 0$ and $f''(\xi) \neq 0$. Further assume there exists $A > 0$ such that*

Note that if $f'(\xi) \neq 0$ then there exist such $A, \delta > 0$.

$$\forall x, y \in I_\delta, \quad \left| \frac{f''(x)}{f'(y)} \right| \leq A.$$

(This implicitly requires $f'(\xi) \neq 0$.) If $|x_0 - \xi| \leq h = \min(\delta, 1/A)$, then Newton's method converges quadratically to ξ.

Proof. Assume $|x_k - \xi| \leq h$ (it is true of x_0, and we will show that if it is true of x_k then it is true of x_{k+1}, so that by induction it will be true of all x_k's.) In particular, $x_k \in I_\delta$ so that we can Taylor expand f around x_k:[2]

$$0 = f(\xi) = f(x_k) + (\xi - x_k) f'(x_k) + \frac{(\xi - x_k)^2}{2} f''(\eta_k)$$

for some η_k between ξ and x_k so that $\eta_k \in I_\delta$. The proof works in two stages. We first show convergence is at least linear; then we show it is actually quadratic.

At least linear convergence. Since $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$, the (signed) error obeys:

$$\begin{aligned} \xi - x_{k+1} &= \xi - x_k + \frac{f(x_k)}{f'(x_k)} \\ &= \frac{f(x_k) + (\xi - x_k) f'(x_k)}{f'(x_k)} && \text{Use Taylor on numerator} \\ &= -\frac{(\xi - x_k)^2}{2} \frac{f''(\eta_k)}{f'(x_k)}. \end{aligned}$$

Using that $x_k, \eta_k \in I_\delta$ and our assumptions,

$$\begin{aligned} |\xi - x_{k+1}| &\leq \frac{1}{2} (\xi - x_k)^2 A && \text{See footnote.}[3] \\ &\leq \frac{1}{2} |\xi - x_k| && \text{Use } |\xi - x_k| A \leq 1 \text{ since } |\xi - x_k| \leq h \leq 1/A. \end{aligned}$$

In particular, $|x_{k+1} - \xi| \leq h$. Since $|x_0 - \xi| \leq h$ by assumption, all x_k satisfy $|x_k - \xi| \leq h$ by induction. Furthermore, x_k converges to ξ at least linearly. Note: we did not yet use $f''(\xi) \neq 0$, but already we get linear convergence.

[2] This is the Lagrange form of the remainder: see Section 1.5.

[3] At this point, it is tempting to go for quadratic convergence directly by studying $\frac{|\xi - x_{k+1}|}{(\xi - x_k)^2}$, but notice that we did not yet prove that $\varepsilon_k = |\xi - x_k|$ converges to zero.

Quadratic convergence. Since $x_k \to \xi$, so does $\eta_k \to \xi$. By continuity,

$$\lim_{k\to\infty} \frac{|x_{k+1} - \xi|}{|x_k - \xi|^2} = \lim_{k\to\infty} \left|\frac{f''(\eta_k)}{2f'(x_k)}\right| = \left|\frac{f''(\xi)}{2f'(\xi)}\right| = \mu > 0.$$

Carefully compare to the definition of quadratic convergence to conclude. □

Question 1.20. *What happens if $f'(\xi) = 0$? Is that good or bad?*

■

Question 1.21. *What happens if $f''(\xi) = 0$? Is that good or bad?*

■

While Newton's method is a great algorithm, bear in mind that the theorem we just established does not provide a practical way of initializing the sequence. This remains a practical issue, which can only be resolved on a case by case basis.

As a remark, note that the convergence guarantees given here are of the form: if initialization is close enough to a root, then we get convergence to that root. It is a common misconception to infer that if there is convergence from a given initialization, then convergence is to the closest root. That is simply not true. See [SM03, §1.7] for illustrations of just how complicated the behavior of Newton's method (and others) can be as a function of initialization.

1.4 Secant method

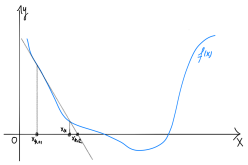
Newton's method is nice, but computing f' can be a pain sometimes. The derivative can even be inaccessible at all for all intents and purposes, if f is given to us not as a mathematical formula but rather as a computer program whose code is either too complicated to dive into, or not revealed to us (a *black box*).

Here is an alternative, assuming f is continuously differentiable. We can approximate the derivative at x_k using the current and the previous iterate:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} = f'(\eta_k)$$

for some η_k between x_k and x_{k-1}. If $x_k \to \xi$, then $|x_k - x_{k-1}| \to 0$ and the approximation gets better. Furthermore, this is cheap because it only relies on quantities that are readily available: the evaluation of f at the two most recent iterates. Plug this into Newton's method to get the *secant method*.

Definition 1.22. *For given x_0, x_1, the* secant method *generates the sequence:*



$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

We implicitly assume that $f(x_k) \neq f(x_{k-1})$ for all k.

With a drawing, convince yourself that x_{k+1} is the root of the line passing through $(x_k, f(x_k))$ and $(x_{k-1}, f(x_{k-1}))$.

Let's try this method on our running example.

```
f  = @(x) exp(x) - 2*x - 1;  % No need for the derivative of f

x0 = 1.5;                    % Need two initial points now
x1 = 1.0;

f0 = f(x0);                  % Evaluate f at both initial points
f1 = f(x1);

fprintf('x = %+.16e, \t f(x) = %+.16e\n', x1, f1);

for k = 1 : 12

    % Compute the next iterate
    x2 = x1 - f1 * (x1-x0) / (f1-f0);

    % Evaluate the function there (single call to f!)
    f2 = f(x2);

    fprintf('x = %+.16e, \t f(x) = %+.16e\n', x2, f2);

    % Slide the window
    % Equivalent code: [x0, f0, x1, f1] = deal(x1, f1, x2, f2);
    x0 = x1;
    f0 = f1;
    x1 = x2;
    f1 = f2;

end
```

This produces the following output:

```
x = +1.0000000000000000e+00,    f(x) = -2.8171817154095447e-01
x = +1.1845136881643570e+00,    f(x) = -9.9307418799988841e-02
x = +1.2859430872139392e+00,    f(x) = +4.6192337895330837e-02
x = +1.2538792881164769e+00,    f(x) = -3.8492759507384733e-03
```

```
x = +1.2563456836075342e+00,    f(x) = -1.2937473932783661e-04
x = +1.2564314625719744e+00,    f(x) = +3.8418517700478105e-07
x = +1.2564312086009526e+00,    f(x) = -3.8149927661379479e-11
x = +1.2564312086261695e+00,    f(x) = -4.4408920985006262e-16
x = +1.2564312086261697e+00,    f(x) = +0.0000000000000000e+00
x = +1.2564312086261697e+00,    f(x) = +0.0000000000000000e+00
x =  NaN,                       f(x) =  NaN
x =  NaN,                       f(x) =  NaN
x =  NaN,                       f(x) =  NaN
```

Convergence to the positive root is very fast indeed, though after getting there things go out of control. Why is that? Propose an appropriate stopping criterion to avoid this situation.

We state a convergence result with a proof sketch here (See [SM03, Thm. 1.10] for details). In [SM03, Ex. 1.10], you are guided to establish superlinear convergence. (This theorem is only for your information.)

Theorem 1.23. *Let f be continuously differentiable on $I = [\xi - h, \xi + h]$ for some $h > 0$ with $f(\xi) = 0, f'(\xi) \neq 0$. If x_0, x_1 are sufficiently close to ξ, then the secant method converges to ξ at least linearly.*

Proof sketch. Assume $f'(\xi) = \alpha > 0$ (the argument is similar for $\alpha < 0$.) In a subinterval I_δ of I, by continuity of f', we have $f'(x) \in [\frac{3}{4}\alpha, \frac{5}{4}\alpha]$. Following the first part of the proof for the convergence rate of Newton's method, this is sufficient to conclude that $|x_{k+1} - \xi| \leq \frac{2}{3}|x_k - \xi|$, leading to at least linear convergence. □

As a closing remark to this chapter, we note that it is a good strategy to use bisection to zoom in on a root at first, thus exploiting the linear convergence rate of bisection and its robustness; then to switch to a superlinearly convergent method such as Newton's or the secant method to "finish the job." This two-stage procedure is part of the strategy implemented in Matlab's `fzero`, as described in a series of blog posts by Matlab creator Cleve Moler.[4]

[4] https://blogs.mathworks.com/cleve/2015/10/12/zeroin-part-1-dekkers-algorithm/, https://blogs.mathworks.com/cleve/2015/10/26/zeroin-part-2-brents-version/, https://blogs.mathworks.com/cleve/2015/11/09/zeroin-part-3-matlab-zero-finder-fzero/

1.5 A quick note about Taylor's theorem

In this course, we frequently use Taylor's theorem with remainder in Lagrange form. The reader is encouraged to consult Wikipedia[5] for a refresher of this useful tool from calculus. For example, at order two the theorem is stated below. We give the classical proof based on Cauchy's mean value theorem (also called the extended mean value theorem).[6] This proof extends to Taylor expansions of any order, provided f is sufficiently many times differentiable.

Theorem 1.24. *Let f be twice differentiable on (x, a) with f' continuous on $[x, a]$. There exists $\eta \in (x, a)$—which depends on a in general—such that*

$$f(a) = f(x) + (a - x)f'(x) + \frac{1}{2}(a - x)^2 f''(\eta).$$

Proof. Consider these functions of t:

$$\begin{aligned} G(t) &= (t - a)^2, & G'(t) &= 2(t - a), \\ F(t) &= f(t) + (a - t)f'(t), & F'(t) &= (a - t)f''(t). \end{aligned}$$

Cauchy's mean value theorem states there exists η (strictly) between a and x such that

$$\frac{F'(\eta)}{G'(\eta)} = \frac{F(x) - F(a)}{G(x) - G(a)}.$$

On one hand, we compute

$$\frac{F(x) - F(a)}{G(x) - G(a)} = \frac{f(x) + (a - x)f'(x) - f(a)}{(x - a)^2}.$$

On the other hand we compute

$$\frac{F'(\eta)}{G'(\eta)} = \frac{(a - \eta)f''(\eta)}{2(\eta - a)} = -\frac{1}{2}f''(\eta).$$

Combine and re-arrange to finish the proof. □

[5] https://en.wikipedia.org/wiki/Taylor%27s_theorem#Explicit_formulas_for_the_remainder

[6] https://en.wikipedia.org/wiki/Mean_value_theorem#Cauchy's_mean_value_theorem

Chapter 2

Inexact arithmetic, IEEE and differentiation

Computers compute inaccurately, in a very precise way.

Vincent Legat, Prof. of Numerical Methods, UCLouvain, 2006

In this chapter, we delve into an important (if frustrating) aspect of computing with real numbers on real computers: it cannot be done exactly.[1] Fortunately, *round off errors*, as they are called, are systematic (as opposed to random) and obey precise rules. We already witnessed an example of math clashing with computation when investigating the bisection algorithm (recall Figure 1.3.) Let's look at another, more important example of this: approximating derivatives.

2.1 Motivating example: finite differentiation

We want to approximate the derivative of a function f, but we are only allowed to compute the value of f itself at some points of our choice. Certainly, computing f at a single point cannot reveal any information about f' (the rate of change.) The next best target is: can we do it with two points?

Problem 2.1. *Let $f\colon \mathbb{R} \to \mathbb{R}$ be three times continuously differentiable in an interval $[x - \bar{h}, x + \bar{h}]$ for some $\bar{h} > 0$. Compute an approximation of $f'(x)$ using only two evaluations of f at well-chosen points.*

[1] Unless we allow an unbounded amount of memory to be used to represent numbers, which is hopelessly impractical.

As often, we start with a Taylor expansion. For any $0 < h < \bar{h}$, there exist η_1 and η_2 both in $[x - \bar{h}, x + \bar{h}]$ such that

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\eta_1),$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\eta_2).$$

Our goal is to obtain a formula for $f'(x)$. Thus, it is tempting to compute the difference between the two formulas above:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}\left(f'''(\eta_1) + f'''(\eta_2)\right).$$

Solving for $f'(x)$, we get:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{12}\left(f'''(\eta_1) + f'''(\eta_2)\right). \tag{2.1}$$

Since f''' is continuous in $[x-\bar{h}, x+\bar{h}]$, it is also bounded in that interval. Let M_3 be such that $|f'''(\eta)| \leq M_3$ for all η in the interval. The approximation

$$f'(x) \approx \Delta_f(x;h) = \frac{f(x+h) - f(x-h)}{2h}$$

is called a *finite difference approximation*. From (2.1), we deduce it incurs an error bounded as:

$$|f'(x) - \Delta_f(x;h)| \leq \frac{M_3}{6}h^2. \tag{2.2}$$

This formula suggests that the smaller h, the smaller the approximation error, which is certainly in line with our intuition about derivatives. Let's verify this on a computer with $f(x) = \sin\left(x + \frac{\pi}{3}\right)$, approximating $f'(0)$.

```
f  = @(x) sin(x + pi/3);
df = @(x) cos(x + pi/3);

% Pick 101 values of h on a log-scale from 1e-16 to 1e0.
hh  = logspace(-16, 0, 101);
err = zeros(size(hh));

for k = 1 : numel(hh)

    h = hh(k);
```

```
    approx = (f(h) - f(-h))/(2*h);
    actual = df(0);

    err(k) = abs(approx - actual);

end

% Bound on |f'''| over the appropriate interval.
% Here, |f'''(x)| = |-cos(x + pi/3)| <= 1 everywhere.
M3 = 1;

loglog(hh, err, '.-', hh, (M3/6)*hh.^2, '-');
legend('Actual error', 'Theoretical bound', 'Location', ...
    'SouthWest');
xlabel('Step size h');
ylabel('Error |f''(0) - FD(f, 0, h)|');
xlim([1e-16, 1]);
ylim([1e-12, 1]);
```

This code generates Figure 2.1. It is pretty clear that when h is "too" small, something breaks. Specifically, it is the fact that our computations are inexact. Fortunately, we will be able to give a precise description of what happens, also allowing us to pick an appropriate value for h in practice.

2.2 A simplified model for IEEE arithmetic

We follow Lecture 13 in [TBI97]: read that first. We assume double precision, which is the default in Matlab.

We are allotted 64 bits to represent real numbers. Let us use one of these bits to code the sign: if the bit is 1, let's agree that the number is nonnegative; if the bit is 0, we agree the number is nonpositive. This concern aside, we have 63 bits left, which allows us to pick $2^{63} \approx 10^{19}$ nonnegative numbers of our choosing. For each possible sequence of 63 bits (each 0 or 1), we get to choose which real number it represents. If we wish to represent a certain real number on our computer, chances are it won't be one of the representable numbers, so we will round it to the nearest representable number. By doing so, we incur a round-off error. Clearly, the spacing between representable numbers is crucial here.

One simple (if naive) strategy that comes to mind is as follows: let us pick some large number $M > 0$, and let us distribute the 2^{63} representable real numbers evenly between 0 and M. A serious drawback of this approach is that, if we want to represent really large numbers, we are forced to take M large, which in turn increases the spacing between any two representable

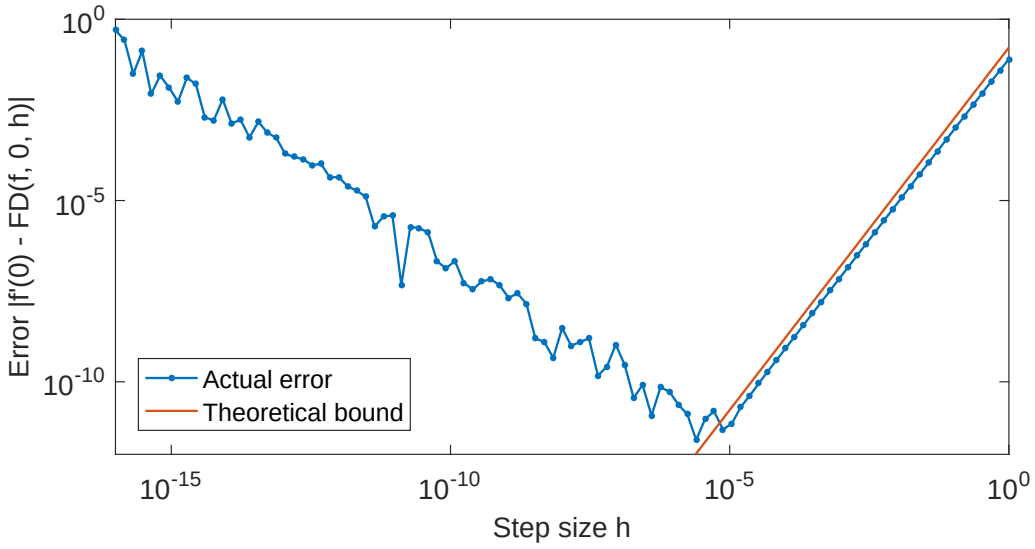


Figure 2.1: Mathematically, we predicted $|f'(x) - \Delta_f(x;h)|$ (the blue curve) should stay below $\frac{M_3}{6}h^2$ (the red line of slope 2). Clearly, something is wrong. From the plot, it seems $h = 10^{-5}$ is a good value. In practice though, we cannot draw this plot for we do not know $f'(x)$: we need to predict what a good h is via other means.

numbers. A related concern is that we are giving the same importance to absolute errors anywhere on the interval: this strategy says it is just as bad to round 10^9 to $10^9 + 1$ as it is to round 1 to $1 + 1$: that is just not true in practice.

A good alternative, close to what modern computers do, is to pick the points on a logarithmic scale. Specifically, for a small value of ε (on the order of 10^{-16} in practice), we pick the numbers that are exactly representable as $1, 1\cdot(1+\varepsilon), 1\cdot(1+\varepsilon)^2, \ldots$, and likewise $1, 1\cdot(1+\varepsilon)^{-1}, 1\cdot(1+\varepsilon)^{-2}, \ldots$ Since ε is very small, at first, the numbers we can represent are very close to 1. But eventually, this being an exponential process, we get to also pick very large and very small numbers. The key is the following: by construction, the relative spacing between two representable numbers is always the same, namely: they are separated by a ratio of $1+\varepsilon$ (very close to 1). The absolute spacing, on the other hand, can grow quite large (or quite small). Indeed, if x is a representable number, then the next representable number is $x\cdot(1+\varepsilon)$. They are separated by an absolute gap of $x\cdot(1+\varepsilon) - x = x\varepsilon$. For $x = 1$, this gap is only on the order of 10^{-16}, but for $x = 10^6$, the gap is much larger: on the order of 10^{-10}. This is acceptable: an error of 10^{-10} on a quantity of 10^6

is not as bad as if we made that error on a quantity on the order of 1. Of course, we still only have a finite number of numbers we can represent, and this simplified explanation also doesn't cover how we represent 0: we leave such concerns aside, as they are not necessary for our purposes.

The IEEE 754 standard codifies a system along the lines described above: this is how (most) computers compute with real numbers, in a setup known as *floating point arithmetic*.[2] This is designed to offer a certain *relative* accuracy over a huge range of numbers. Ignoring overflow and underflow problems[3] as well as denormalized numbers[4] (which we will always do), the main points to remember are:

1. Real numbers are rounded to representable numbers (on 64 bits) with relative accuracy $\varepsilon_{\text{mach}} = 1.11 \cdot 10^{-16}$.
2. For individual, basic operations, such as $+, -, \cdot, /, \sqrt{}$ (but also, on modern computers, special functions such as trigonometric functions, exponentials...) *on representable numbers*, results of one operation are *as accurate as can be*, in that the result is the representable number which is closest to the correct answer.

Thus, for a given real number a, its representation $fl(a)$ in double precision obeys[5]

$$fl(a) = a(1 + \varepsilon_0), \text{ with } |\varepsilon_0| \leq \varepsilon_{\text{mach}}.$$

Furthermore, given two numbers a and b *already represented exactly* in memory (that is, $fl(a) = a, fl(b) = b$), we can assume the following about operations with these numbers:

- $a \oplus b = fl(a + b) = (a + b)(1 + \varepsilon_1)$,
- $a \ominus b = fl(a - b) = (a - b)(1 + \varepsilon_2)$,
- $a \odot b = fl(ab) = ab(1 + \varepsilon_3)$,

[2] https://en.wikipedia.org/wiki/IEEE_floating_point

[3] Overflow occurs when one attempts to work with a number larger than the biggest number which can be stored (about 10^{308}); underflow occurs when one attempts to store a number which is closer to zero than the closest nonzero number which can be represented.

[4] Denormalized numbers fill the gap around zero to improve accuracy there, but the relative accuracy is not as good as everywhere else.

[5] Remark that $\varepsilon_{\text{mach}}$ is half of ε above, since if a is in the interval $[x, x(1 + \varepsilon)]$ whose limits are exactly represented, its distance to either limit is at most half of the interval length, that is, $\frac{\varepsilon}{2}x$. Then, the relative error upon rounding a to its closest representable number is $\frac{|a - fl(a)|}{|a|} \leq \frac{\varepsilon}{2}\frac{|x|}{|a|} \leq \frac{\varepsilon}{2} \triangleq \varepsilon_{\text{mach}}$.

- $a \oslash b = fl(a/b) = \frac{a}{b}(1+\varepsilon_4)$,
- $\text{sqrt}(a) = \sqrt{a}(1+\varepsilon_5)$,

where $|\varepsilon_i| \leq \varepsilon_{\text{mach}}$ for all i.

Finally, we add as an extra rule that

Multiplication and division by 2 is exact.

This follows from the fact that computers typically represent numbers in binary, so that multiplying and dividing by 2 can be done exactly. Consequently, powers of 2 (and of 1/2) are exactly representable, and multiplying or dividing by them is done exactly. Similarly, since we typically use one bit to encode the sign,

> If a can be represented exactly, then the same is true of $-a$. Hence, computing $-a$ is error-free.

Computations usually involve many simple operations executed in succession, so that round-off errors will combine. Let's see how addition of three numbers works out (notice that we now need to specify the order in which additions are computed):

$$\begin{aligned} a \oplus (b \oplus c) &= a \oplus (b+c)(1+\varepsilon_1) \\ &= [a + (b+c)(1+\varepsilon_1)]\,(1+\varepsilon_2) \\ &= (a+b+c) + \varepsilon_2(a+b+c) + \varepsilon_1(b+c) + \varepsilon_1\varepsilon_2(b+c) \\ &= (a+b+c) + \varepsilon_2(a+b+c) + \varepsilon_1(b+c) + O(\varepsilon_{\text{mach}}^2). \end{aligned}$$

In the last equation, we made a simplification which we will always do: terms proportional to $\varepsilon_{\text{mach}}^2$ (or $\varepsilon_{\text{mach}}^3, \varepsilon_{\text{mach}}^4 \ldots$) are so small that we do not care; so we hide them in the notation $O(\varepsilon_{\text{mach}}^2)$. Notice how the formula tells us the result of the addition (after both round-offs) is equal to the correct sum $a+b+c$, plus some extra terms. It is useful to bound the error:

$$|a \oplus (b \oplus c) - (a+b+c)| \leq \varepsilon_{\text{mach}}\,(|a+b+c| + |b+c|) + O(\varepsilon_{\text{mach}}^2).$$

In relative terms, we get

$$\left| \frac{a \oplus (b \oplus c) - (a+b+c)}{a+b+c} \right| \leq \varepsilon_{\text{mach}} \left(1 + \frac{|b+c|}{|a+b+c|} \right) + O(\varepsilon_{\text{mach}}^2).$$

Question 2.2. *Is there a preferred order in which to sum a, b, c to reduce the error? Think of $a \ll b \ll c$.*

■

Question 2.3. *Can you establish a formula for the sum of* $a_1, \ldots, a_n$*?*

■

Matlab's `eps` function gives the spacing between a representable number and the next representable number:

```
>> help eps
 eps  Spacing of floating point numbers.
D = eps(X), is the positive distance from ABS(X) to the next
larger in magnitude floating point number of the same precision
as X.
```

Essentially, `eps(x)` is $2|x|\varepsilon_{\text{mach}}$. Notice the phrase "of the same precision as X": this is because Matlab allows computations in both double precision (as described above), and in single precision. In single precision, only 32 bits are used to represent real numbers, and $\varepsilon_{\text{mach}} \approx 6 \cdot 10^{-8}$: `eps(single(1))/2`. Sometimes, the reduced computation time is more important than the resulting loss of accuracy.

Below, we are going to find out that the main issue with the finite differentiation example is the computation of $a - b$ where $|a|, |b|$ are large yet $a \approx b$. To see why that is, consider computing $fl(a - b)$ in a situation where a, b are *not* exactly represented. Then:

$$\begin{aligned} fl(a-b) &= fl(a) \ominus fl(b) = (a(1+\varepsilon_1) - b(1+\varepsilon_2))(1+\varepsilon_3) \\ &= (a-b)(1+\varepsilon_3) + \varepsilon_1 a - \varepsilon_2 b + O(\varepsilon_{\text{mach}}^2). \end{aligned}$$

In terms of relative accuracy, we get

$$\left| \frac{fl(a-b) - (a-b)}{a-b} \right| \leq \varepsilon_{\text{mach}} \left(1 + \frac{|a| + |b|}{|a-b|} \right) + O(\varepsilon_{\text{mach}}^2).$$

Clearly, if $|a - b| \ll |a| + |b|$, we are in trouble. This happens if a, b are large yet close, so that rounding them individually incurs errors that are small relative to themselves, yet large relative to our actual target: their difference.

2.3 Finite differentiation

Formula (2.2) is mathematically correct: it gives the so-called *truncation error* of the finite difference approximation. Yet, as illustrated by Figure 2.1,

it is reckless to rely on it in IEEE arithmetic. Why? The short answer is: when h is very small, $f(x+h)$ and $f(x-h)$ are almost equal (to $f(x)$). Thus, their difference is much smaller in magnitude than their own values in magnitude. But, following the IEEE system, each of $f(x-h)$ and $f(x+h)$ is stored with a relative accuracy proportional to (essentially) $|f(x)|$. Thus, the difference is computed with an error proportional to $|f(x)|$ as well, *not* proportional to the difference. So, for small h, the error might be much larger than the value we are computing!

We effectively compute $fl(\Delta_f(x;h))$. How much error (overall) do we incur for that? The classic trick is to start with a triangular inequality, so that we can separate on one side the round-off error, and on the other side the truncation error (that is, the mathematical error):

$$\begin{aligned}|fl(\Delta_f(x;h)) - f'(x)| &= |fl(\Delta_f(x;h)) - \Delta_f(x;h) + \Delta_f(x;h) - f'(x)| \\ &\leq |fl(\Delta_f(x;h)) - \Delta_f(x;h)| + |\Delta_f(x;h) - f'(x)|\,.\end{aligned}$$

The truncation error we already understand: it is bounded by $\frac{M_3}{6}h^2$. Let's focus on the round-off error. For notational convenience, we let $x = 0$ and omit it in the notation. We also assume f can be evaluated with relative accuracy ε at $x \pm h$:

$$fl(f(x \pm h)) = f(x \pm h)(1 + \varepsilon_{1,2}), \qquad \text{with } |\varepsilon_{1,2}| \leq \varepsilon_{\text{mach}}.$$

We could say $10\varepsilon_{\text{mach}}$ or $100\varepsilon_{\text{mach}}$: that is not what matters here. And since we get to pick h, we might as well pick one that is represented exactly (and $2h$ will be too).[6]

$$\begin{aligned}fl(\Delta_f(h)) &= (fl(f(h)) \ominus fl(f(-h))) \oslash (2h) \\ &= \frac{f(h)(1+\varepsilon_1) - f(-h)(1+\varepsilon_2)}{2h}(1+\varepsilon_3)(1+\varepsilon_4).\end{aligned}$$

Let's break it down: $\varepsilon_1, \varepsilon_2$ are the relative errors due to computing $f(h)$ and $f(-h)$; ε_3 is the error incurred by computing their difference; and ε_4 is the error due to the division.[7] Let us reorganize terms to make $\Delta_f(h)$ appear:

$$\begin{aligned}fl(\Delta_f(h)) &= \frac{f(h)(1+\varepsilon_1) - f(-h)(1+\varepsilon_2)}{2h}(1+\varepsilon_3)(1+\varepsilon_4) \\ &= \frac{f(h) - f(-h)}{2h}(1+\varepsilon_3)(1+\varepsilon_4) + \frac{\varepsilon_1 f(h) - \varepsilon_2 f(-h)}{2h}(1+\varepsilon_3)(1+\varepsilon_4) \\ &= \Delta_f(h) + \Delta_f(h)(\varepsilon_3 + \varepsilon_4) + \frac{\varepsilon_1 f(h) - \varepsilon_2 f(-h)}{2h} + O(\varepsilon^2).\end{aligned}$$

[6] If h is not exactly represented, we get an error in the denominator which appears as $\frac{1}{1+\varepsilon_5}$. It is then useful to recall that, since $|\varepsilon_5| \leq \varepsilon_{\text{mach}}$, we have $\frac{1}{1+\varepsilon_5} = 1 - \varepsilon_5 + O(\varepsilon_{\text{mach}}^2)$.

[7] You can even get rid of ε_4 if you restrict youself to h being a power of $1/2$.

A number of terms had a product of two or more ε_i's in them; we hid them all under $O(\varepsilon^2)$. Do this soon to simplify computations. So, the round-off error is made of three terms:

$$|fl(\Delta_f(h)) - \Delta_f(h)| \leq 2\varepsilon|\Delta_f(h)| + \left|\frac{\varepsilon_1 f(h) - \varepsilon_2 f(-h)}{2h}\right| + O(\varepsilon^2).$$

The first term is fine: it is the usual form for a relative error. The last term is also fine: we consider $O(\varepsilon^2)$ very small. It is the middle term which is the culprit. To understand why it is harmful, recall that ε_1 and ε_2 can be both positive and negative (corresponding to rounding up or down when the operation was computed.) Thus, if the signs of $\varepsilon_1 f(h)$ and $\varepsilon_2 f(-h)$ happen to be opposite (which might very well be the case), then the numerator is quite large. Using $f(\pm h) = f(0) \pm hf'(0) + O(h^2)$:

$$\left|\frac{\varepsilon_1 f(h) - \varepsilon_2 f(-h)}{2h}\right| \leq \frac{\varepsilon|f(h)| + \varepsilon|f(-h)|}{2h} \leq \varepsilon\frac{|f(0)|}{h} + \varepsilon|f'(0)| + O(\varepsilon h).$$

Clearly, if h is small, this is bad. Overall then, we find the following round-off error (where we also used $|\Delta_f(h) - f'(0)| = O(h^2)$):

$$|fl(\Delta_f(h)) - \Delta_f(h)| \leq 3\varepsilon|f'(0)| + \varepsilon\frac{|f(0)|}{h} + O(\varepsilon^2) + O(\varepsilon h).$$

Finally, we have the following formula to bound the error; at this point, we re-integrate x in our notation:

$$|fl(\Delta_f(x;h)) - f'(x)| \leq \frac{M_3}{6}h^2 + 3\varepsilon|f'(x)| + \varepsilon\frac{|f(x)|}{h} + O(\varepsilon^2) + O(\varepsilon h). \tag{2.3}$$

Good. This should help us pick a suitable value of h. The goal is to minimize the parts of the bound that depend on h. This is pretty much the case when both error terms are equal:[8]

$$\frac{M_3}{6}h^2 \approx \varepsilon\frac{|f(x)|}{h},$$

thus, $h = \sqrt[3]{\frac{6|f(x)|}{M_3}\varepsilon}$ is an appropriate choice. If the constant is not too different from 1, the magic value to remember is $h \approx \sqrt[3]{\varepsilon} \approx 10^{-5}$.

[8]More precisely, you could observe that $\frac{M_3}{6}h^2 + \varepsilon\frac{|f(x)|}{h}$ attains its minimum when the derivative with respect to h is zero; this happens for $h = \sqrt[3]{\frac{3|f(x)|}{M_3}\varepsilon}$.

Question 2.4. *Looking at the culprit in the error bound,* $\varepsilon\frac{|f(0)|}{h}$*, one might think that it is sufficient to work with* $g(x) \equiv f(x) - f(0)$ *instead, so that* $g'(0) = f'(0)$*, and the culprit does not appear when computing* $g'(0)$ *since* $g(0) = 0$*. Why is that a fallacy?*

■

Question 2.5. *Can you track down what happens if the computation of* $f(h)$ *and* $f(-h)$ *only has a relative accuracy of* $100\varepsilon_{\text{mach}}$ *instead of* $\varepsilon_{\text{mach}}$*? (Follow* ε_1 *and* ε_2*.)*

■

The following bit of code adds the IEEE-aware error bound to the mix, as depicted in Figure 2.2.

```
hold all;
loglog(hh, (M3/6)*hh.^2 + 3*eps(1)*abs(df(0)) + ...
    (eps(1)./hh)*abs(f(0)));
legend('Actual error', 'Theoretical bound', 'IEEE-aware bound');
```

Question 2.6. *Notice how, in this log-log plot, the right bit has a slope of 2, whereas the left bit has a slope of* -1*. Can you tell why? The precise value of the optimal* h *depends on* M_3 *which is typically unknown. Is it better to overestimate or underestimate?*

■

2.4 Bisection

Recall the bisection method: for a certain continuous function f, let $a_0 < b_0$ (represented exactly in memory) be the end points of an interval such that $f(a_0)f(b_0) < 0$ (so the interval contains a root). The bisection method computes $c_0 = \frac{a_0+b_0}{2}$, the mid-point of the interval $[a_0, b_0]$, and decides to make the next interval either $[a_1, b_1] = [a_0, c_0]$ or $[a_1, b_1] = [c_0, b_0]$. Then, it iterates. If this is done exactly, the interval length

$$\ell_k = b_k - a_k$$

is reduced by a factor of two at each iteration, so that $\ell_k = \frac{1}{2^k}\ell_0 \to 0$. Ah, but computations are not exact...

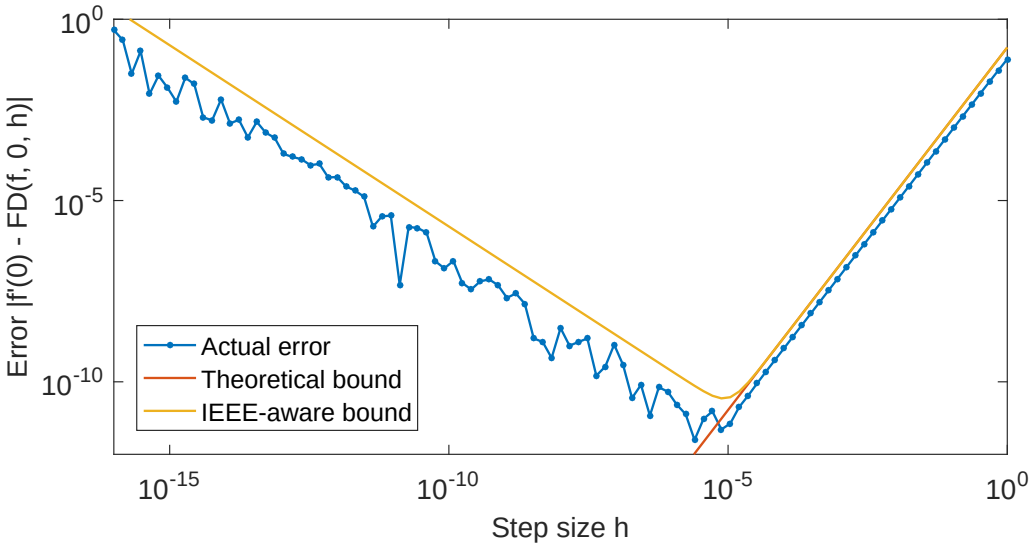


Figure 2.2: Factoring in round-off error in our analysis, we get a precise understanding of how accurately the finite difference formula can approximate derivatives in practice. The rule of thumb $h \approx 10^{-5}$ is fine in this instance.

What happens if we iterate until ℓ_k becomes very small (think: small enough that machine precision becomes an issue)? We eventually get to the point where a_k and b_k are "next to each other" in the list of exactly representable numbers. Thus, when computing c_k, which should fall in between the two, we instead get $fl(c_k)$ rounded to either a_k or b_k: the interval will no longer change, and the interval length will no longer decrease. How long is the interval at that point? About $\varepsilon_{\text{mach}}|a_k|$. Since at convergence a_k, b_k should have converged to bracket a root ξ, we may expect a good bound to be $\ell_k \lessapprox \frac{1}{2^k}\ell_0 + \varepsilon_{\text{mach}}|\xi|$.[9] Indeed, Figure 2.3 shows Figure 1.3 with an added IEEE-aware bound which explains the behavior exactly.

The following piece of code confirms the explanation, by verifying that once bisection gets stuck, a_k, b_k are indeed "neighbors" in the finite set of representable reals.

```
% run bisection; then:
fprintf('a = %.16e\n', a);
fprintf('b = %.16e\n', b);
```

[9]The bound is only approximate because the interval is not quite exactly halved at each iteration, also because of round-off errors. That effect is negligible, but it is a good exercise to try and account for it.

```
fprintf('c = %.16e\n', c);
fprintf('b - a  = %.16e\n', b - a);
fprintf('eps(a) = %.16e\n', eps(a));
```

The output is:

```
a = 4.9651142317442760e+00
b = 4.9651142317442769e+00
c = 4.9651142317442769e+00
b - a  = 8.8817841970012523e-16
eps(a) = 8.8817841970012523e-16
```

Indeed, the distance between a and the next representable number as given by `eps(a)` is exactly the distance between a and b. As a result, c is rounded to either a or b (in this case, to b.)

A final remark: the story above suggests we can get an approximation of a root ξ basically up to machine precision. If you feel courageous, you could challenge this statement and ask: how about errors in computing $f(c_k)$? When c_k is close to a root, $f(c_k)$ is close to zero, hence we might get the sign wrong and move to the wrong half interval... We won't go there today.

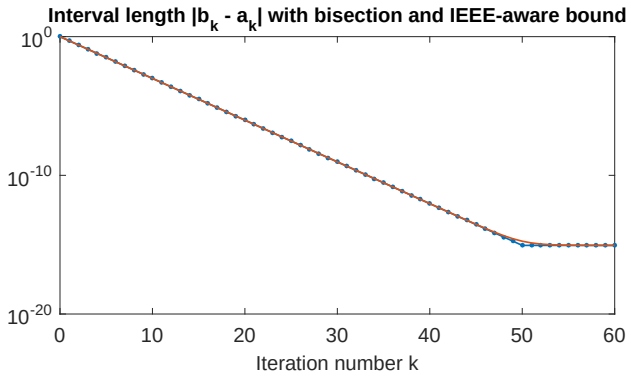


Figure 2.3: Figure 1.3 with an extra curve: the red curve shows the bound $\frac{1}{2^k}\ell_0 + \varepsilon_{\text{mach}}|\xi|$ which better predicts the behavior of the interval length during bisection under inexact arithmetic.

2.5 Computing long sums

How accurately can we compute $\sum_{i=1}^{n} \frac{1}{i^2}$? There is a preferred order. To understand it, we need a formula controlling the round-off error incurred in

computing a sum of n numbers $x_1, \ldots, x_n$, where $x_i = 1/i^2$ in our case. We will not assume that x_i is represented exactly in memory. Thus,

$$fl\left(\sum_{i=1}^{n} x_i\right) = fl(x_1) \oplus \cdots \oplus fl(x_n) = \bigoplus_{i=1}^{n} fl(x_i),$$

where we have to specify the order of summation. Let's say we sum x_1 with x_2, then the result with x_3, then the result with x_4, etc.—that is, we sum the big numbers first. Using $fl(x_i) = x_i(1+\varepsilon_i)$, establish the following identity, where $\varepsilon_{(i)}$ is the relative error incurred by the ith addition (there are $n-1$ of them):

$$\bigoplus_{i=1}^{n} fl(x_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} x_i \left(\varepsilon_i + \sum_{j=i-1}^{n-1} \varepsilon_{(j)}\right) + O(\varepsilon_{\text{mach}}^2).$$

(We tacitly defined $\varepsilon_{(0)} = 0$ for ease of notation.)

For the sum we are interested in, x_i can be computed as $1/i/i$ or $1/(i \cdot i)$, thus involving two basic operations. This means that, up to $O(\varepsilon_{\text{mach}}^2)$ terms, we can bound $|\varepsilon_i| \le 2\varepsilon$. Plugging this into the above identity, we get the following error bound:

$$\begin{aligned}\left|\sum_{i=1}^{n} \frac{1}{i^2} - \bigoplus_{i=1}^{n} fl(1/i/i)\right| &\le \sum_{i=1}^{n} \frac{2 + (n-1) - (i-1) + 1}{i^2} \varepsilon_{\text{mach}} + O(\varepsilon_{\text{mach}}^2) \\ &= 3\varepsilon_{\text{mach}} \sum_{i=1}^{n} \frac{1}{i^2} + \varepsilon_{\text{mach}} \sum_{i=1}^{n} \frac{n-i}{i^2} + O(\varepsilon_{\text{mach}}^2). \quad (2.4)\end{aligned}$$

Since $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$ and $\log(n) \le \sum_{i=1}^{n} \frac{1}{i} \le 1 + \log(n)$, we further get

$$\left|\sum_{i=1}^{n} \frac{1}{i^2} - \bigoplus_{i=1}^{n} fl(1/i/i)\right| \le \left[(3+n)\frac{\pi^2}{6} - \log(n)\right] \varepsilon_{\text{mach}} + O(\varepsilon_{\text{mach}}^2).$$

This bounds the error to roughly $\frac{\pi^2}{6} n \varepsilon_{\text{mach}}$, which, for large n, is a relative error of about $n\varepsilon_{\text{mach}}$. This is not so good: if $n = 10^9$, then we only expect 6 accurate digits after the decimal point.

On the other hand, if we had summed with i ranging from n to 1 rather than from 1 to n—small numbers first—then in (2.4) we would have summed $i/i^2 = 1/i$ rather than $(n-i)/i^2$, so that

$$\begin{aligned}\left|\sum_{i=1}^{n} \frac{1}{i^2} - \bigoplus_{i=1}^{n} fl(1/i/i)\right| &\le 4\varepsilon_{\text{mach}} \sum_{i=1}^{n} \frac{1}{i^2} + \varepsilon_{\text{mach}} \sum_{i=1}^{n} \frac{1}{i} + O(\varepsilon_{\text{mach}}^2) \\ &\le \left[4\frac{\pi^2}{6} + \log(n) + 1\right] \varepsilon_{\text{mach}} + O(\varepsilon_{\text{mach}}^2).\end{aligned}$$

This is much better! For $n = 10^9$, the error is smaller than $29\varepsilon_{\text{mach}} < \frac{1}{2}10^{-14}$, which means the result is accurate up to 14 digits after the decimal point!

One final point: in experimenting with this, be careful that even though $\sum_{i=1}^{n} \frac{1}{i^2} \to \frac{\pi^2}{6}$ as $n \to \infty$, for finite n, the difference may be bigger than 10^{-14}. In particular, if we let $n = 10^9$, then

$$\frac{\pi^2}{6} - \sum_{i=1}^{n} \frac{1}{i^2} = \sum_{i=10^9+1}^{\infty} \frac{1}{i^2} \geq \sum_{i=10^9+1}^{2\cdot 10^9} \frac{1}{i^2} \geq 10^9 \frac{1}{(2 \cdot 10^9)^2} = .25 \cdot 10^{-9}.$$

Thus, when comparing the finite sum with $\frac{\pi^2}{6}$, at best, only 9 digits after the decimal point will coincide (and it could be fewer); that is not a mistake: it only has to do with the convergence of that particular sum.

```
n = 1e10;   % this takes a while to run

total1 = 0;
for ii = 1:1:n
    total1 = total1 + 1/ii^2;
end
fprintf('Sum large first:  %.16f\n', total1);

total2 = 0;
for ii = n:-1:1
    total2 = total2 + 1/ii^2;
end
fprintf('Sum small first:  %.16f\n', total2);

fprintf('Asymptotic value: %.16f\n', pi^2 / 6);

% That's about 6 accurate digits and we got 7.
fprintf('Without being careful, we expect an error of about ...
   %.2e\n', n*eps(pi^2/6));
```

```
Sum large first:  1.6449340578345750  % (8th digit is off.)
Sum small first:  1.6449340667482264  % (10th digit is off.)
Asymptotic value: 1.6449340668482264
Without being careful, we expect an error of about 2.22e-06
```

Question 2.7. *At a "big picture level", why does it make sense to sum the small numbers first?*

■

Chapter 3

Linear systems of equations

In this chapter, we solve linear systems of equations ($Ax = b$), we discuss the sensitivity of the solution x to perturbations in b, we consider the implications for solving least-squares problems, and get into the important problem of computing QR factorizations. As we do so, we encounter a number of algorithms for which we ask the questions: What is its complexity in flops (floating point operations)? And also: What could break the algorithm?

We follow mostly the contents of [TBI97]: specific lectures are referenced below. See blackboard for Matlab codes used in class.

3.1 Solving $Ax = b$

We aim to solve the following problem, where we assume A is invertible to ensure existence and uniqueness of the solution.

Problem 3.1 (System of linear equations)**.** *Given an invertible matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, find $x \in \mathbb{R}^n$ such that $Ax = b$.*

At first, we consider a particular class of that problem.

Problem 3.2 (Triangular system of linear equations)**.** *Given an invertible* upper triangular *matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, find $x \in \mathbb{R}^n$ such that $Ax = b$.*

An upper triangular matrix A obeys $a_{ij} = 0$ if $i > j$. For a 6×6 matrix, this is the following pattern, where $\times$ indicates an entry which may or may

not be zero, while other entries are zero:

$$A = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix}.$$

Question 3.3. *Prove that A upper triangular is invertible if and only if $a_{kk} \neq 0$ for all k.*

■

Triangular systems are particularly simple to solve: solve first for x_n, then work your way up obtaining x_{n-1}, x_{n-2} up to x_1. This is called *back substitution*: see first two pages of Lecture 17 in [TBI97] (the remainder of that lecture concerns the stability of the algorithm, which we do not cover in class.)

Algorithm 3.1 Back substitution (Trefethen and Bau, Alg. 17.1)

1: Given: $A \in \mathbb{R}^{n\times n}$ upper triangular and $b \in \mathbb{R}^n$
2: **for** $k = n, n-1, \ldots, 1$ **do** ▷ Work backwards
3: $\quad x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj} x_j}{a_{kk}}$
4: **end for**

Question 3.4. *What is the complexity of this algorithm in flops, that is: how many floating point operations (or arithmetic operations) are required to execute it, as a function of n?*

■

Question 3.5. *Assuming exact arithmetic, could this algorithm break?*

■

Now that we know how to solve triangular systems, a nice observation is that if A is not necessarily triangular itself but we can somehow factorize it into a product $A = LU$, where L is lower triangular and U is upper triangular, then solving $Ax = b$ is also easy. Indeed:

$$Ax = b \qquad \equiv \qquad LUx = b \qquad \equiv \qquad \begin{cases} Ly = b \\ Ux = y. \end{cases}$$

Algorithm 3.2 LU factorization without pivoting [TBI97, Alg. 20.1]

$U \leftarrow A$, $L \leftarrow I$
for k in $1 : (n-1)$ **do** ▷ For each diagonal entry
  for j in $(k+1) : n$ **do** ▷ For each row below row k
    $\ell_{jk} \leftarrow \frac{u_{jk}}{u_{kk}}$ ▷ 1 divide
    $u_{j,k:n} \leftarrow u_{j,k:n} - \ell_{jk} u_{k,k:n}$ ▷ $n-k$ multiply and $n-k$ subtract
  end for
end for

Algorithm 3.3 LU factorization with partial pivoting [TBI97, Alg. 21.1]

$U \leftarrow A$, $L \leftarrow I$, $P \leftarrow I$
for k in $1 : (n-1)$ **do** ▷ For each diagonal entry
  Select $i \geq k$ such that $|u_{ik}|$ is maximized ▷ Pivot selection
  $u_{k,k:n} \leftrightarrow u_{i,k:n}$ ▷ Row swap effect
  $\ell_{k,1:(k-1)} \leftrightarrow \ell_{i,1:(k-1)}$ ▷ Row swap effect
  $p_{k,:} \leftrightarrow p_{i,:}$ ▷ Row swap effect
  for j in $(k+1) : n$ **do** ▷ For each row below row k, do elimination
    $\ell_{jk} \leftarrow \frac{u_{jk}}{u_{kk}}$
    $u_{j,k:n} \leftarrow u_{j,k:n} - \ell_{jk} u_{k,k:n}$
  end for
end for

Thus, we can first solve $Ly = b$ (L is lower triangular: you should be able to adapt the back substitution algorithm to this case easily), then solve $Ux = y$. If A is invertible, then L, U are both invertible (why?). Thus, both back substitutions are valid (at least, in exact arithmetic). In fact, we can make L *unit* lower triangular, which means $\ell_{kk} = 1$ for all k. As a result, there are no divisions involved in back substitution with L and the complexity of that part is $n^2 - n$ flops instead of n^2.

It turns out you (most likely) already learned the algorithm to factor $A = LU$ in your introductory linear algebra class, likely under the name Gaussian Elimination (and the fact it results in an LU factorization was probably not highlighted.) See Lectures 20 and 21 in [TBI97] for Gaussian elimination, with and without pivoting: we cover these in class. See Algorithms 3.2 and 3.3.

With pivoting, we get a factorization of the form

$$PA = LU,$$

where L is unit lower triangular, U is upper triangular and P is a permutation matrix. Given this factorization, systems $Ax = b$ can be solved in exactly

$n^2 - n$ flops still, since the presence of a permutation matrix requires no additional arithmetic operations:

$$Ax = b \quad \equiv \quad PAx = Pb \quad \equiv \quad LUx = Pb \quad \equiv \quad \begin{cases} Ly &= Pb \\ Ux &= y. \end{cases}$$

In these lectures, you find that computing P, L, U requires $\sim \frac{2}{3}n^3$ flops.

The total cost of solving $LUx = Pb$ is thus exactly $2n^2 - n$ flops, which is the same as the cost of computing a matrix-vector product Ax (verify this). In other terms: if we obtain a factorization $PA = LU$, then solving linear systems in A becomes as cheap and easy to do as to compute the product $A^{-1}b$ if A^{-1} were available. Then one may wonder: is there any advantage to computing an LU factorization as opposed to computing A^{-1}? To answer this, we should first ask: how would we compute A^{-1}? It turns out that a good algorithm for this is simply to use an LU factorization of A to solve the n linear systems $Ax = e_i$ for $i = 1, \ldots, n$, where e_i is the ith column of the identity matrix (this is what Matlab does).[1] Indeed, the solutions of these linear systems are the columns of A^{-1} (why?). Using this method, computing A^{-1} involves an LU factorization ($\sim \frac{2}{3}n^3$ flops) and n system solves ($\sim 2n^3$ flops), for a total of $\sim \frac{8}{3}n^3$ flops. Then, we still need to compute the product $A^{-1}b$ for an additional $\sim 2n^2$ flops. This is counter-productive: it would have been much better to use the LU factorization to solve $Ax = b$ directly. An important take away from this is: even if you need to "apply A^{-1} repeatedly", *do not* compute A^{-1} explicitly. Rather, compute the LU factorization of A and use back-and-forward substitution. This is cheaper (because there was no need to compute A^{-1}) and incurs less round-off error simply because it involves far fewer computations. This explanation explains this recommendation in Matlab's documentation: "It is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations `Ax = b`. One way to solve the equation is with `x = inv(A)*b`. A better way, from the standpoint of both execution time and numerical accuracy, is to use the matrix backslash operator `x = A\b`. This produces the solution using Gaussian elimination, without explicitly forming the inverse." Thus, Matlab's backslash operator internally forms the LU factorization. If we need to solve many systems involving the same matrix A and not all the right-hand sides are known at the same time, then it is likely better to use Matlab's `lu` function to save the factorization, then use that—more in Precept 3.

[1] From Matlab's documentation: "`inv` performs an LU decomposition of the input matrix (or an LDL decomposition if the input matrix is Hermitian). It then uses the results to form a linear system whose solution is the matrix inverse `inv(X)`."

After studying these lectures, take a moment to engrave the following in your memory: even if A is invertible and exact arithmetic is used, Gaussian elimination without pivoting may fail (in fact, the factorization $A = LU$ may not exist.) Here is an example to that effect:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Gaussian elimination without pivoting fails at the first step. In contrast, Gaussian elimination with pivoting produces a factorization $PA = LU$ without trouble in this case. Furthermore, following the general principle that *if something is mathematically impossible at 0, it will be numerically troublesome near 0*, it is true that LU decomposition without pivoting is possible for

$$A = \begin{pmatrix} \frac{1}{3}\varepsilon_{\text{mach}} & 1 \\ 1 & 1 \end{pmatrix},$$

but doing so in inexact arithmetic results in a large error $A - LU = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$. Here too, pivoting fixes the issue efficiently.

In three words: pivoting is not just a good idea; it is *necessary in general*. Interestingly, there are certain special cases where pivoting is not necessary; see for example Lecture 23 (Cholesky factorization).

3.2 Conditioning of $Ax = b$

See Lecture 12 in [TBI97] for general context. We discuss mainly the following points in class.

When solving $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ invertible and b nonzero, how *sensitive* is the solution x to perturbations in the right hand side b? If b is perturbed and becomes $b + \delta b$ for some $\delta b \in \mathbb{R}^n$, then surely the solution of the linear system changes as well, and becomes:

$$A(x + \delta x) = b + \delta b. \tag{3.1}$$

How large could the deviation δx be? We phrase this question in relative terms, that is, we want to bound the relative deviation in terms of the relative perturbation:

$$\frac{\|\delta x\|}{\|x\|} \leq [\text{ something to determine }] \frac{\|\delta b\|}{\|b\|}.$$

In asking this question, we want to consider the worst case over all possible perturbations b. It is also important to stress that this is about the sensitivity of the problem, not about the sensitivity (or rather, stability) of any given algorithm. If a problem is sensitive to perturbations, this affects all possible algorithms to solve that problem. We must keep this in mind when assessing candidate algorithms *("à l'impossible, nul n'est tenu.")*

One concept is particularly important to our discussion: the notion of *matrix norm*—see Lecture 3 in [TBI97]. Given a vector norm $\|\cdot\|$ (for example, $\|x\|_2 = \sqrt{x^T x}$), we can define a *subordinate matrix norm* as

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}.$$

Intuitively, this is the largest factor by which A can change the norm of any given vector.

Question 3.6. *Show that for the 2-norm, $\|A\|_2 = \sigma_{\max}(A)$, the largest singular value of A.*

■

Going back to (3.1), since $Ax = b$, we have $A\delta x = \delta b$. In bounding $\frac{\|\delta x\|}{\|x\|}$, one task is to upper bound $\|\delta x\|$; using the definition of subordinate matrix norm:

$$\|\delta x\| = \|A^{-1}\delta b\| \leq \|A^{-1}\|\|\delta b\|.$$

On the other hand, we must lower bound $\|x\|$; since $Ax = b$, we have

$$\|b\| = \|Ax\| \leq \|A\|\|x\|,$$

hence,

$$\|x\| \geq \frac{\|b\|}{\|A\|}.$$

Combining, we get:

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|},$$

which is of the required form. We call the multiplicative factor

$$\kappa(A) = \|A\|\|A^{-1}\| \tag{3.2}$$

the *condition number of A* with respect to the norm $\|\cdot\|$. It is always at least 1,[2] and is infinite if A is not invertible.

[2] Because $\|AB\| \leq \|A\|\|B\|$ for subordinate norms [TBI97, eq. (3.14)]; hence, $1 = \|I\| = \|AA^{-1}\| \leq \|A\|\|A^{-1}\|$.

Question 3.7. *Show that for the 2-norm,* $\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$.

■

In the 2-norm, the situation is particularly clear: A is invertible if and only if $\sigma_{\min}(A) \neq 0$; The notion of conditioning says: if $\sigma_{\min}(A)$ is close to zero *relative to* $\sigma_{\max}(A)$, then even though A is invertible, solving linear systems in A will be tricky because the solution x could be very sensitive to perturbations of b (if those perturbations happen to align with the worst directions.) When $\kappa(A)$ is much larger than 1, we say A is *ill-conditioned*, and we say that solving $Ax = b$ is an ill-conditioned problem.

Material in [TBI97, Lecture 12] shows different matrix computation problems whose conditioning is also given by $\kappa(A)$, which is why we often omit to distinguish between conditioning of the problem at hand and the condition number of A.

As to the origin of perturbations δb, they can of course come from the source of the data: after all, b typically is the result of some experiment, and as such it is usually of finite accuracy. But of course, even if b is known exactly, the simple act of expressing it as floating point numbers in the IEEE standard results in a relative error on the order of $\varepsilon_{\text{mach}}$. Thus, $\frac{\|\delta b\|}{\|b\|}$ should be expected to be always at least of that order.

A rule of thumb is that in solving $Ax = b$ in inexact arithmetic, assuming perfect knowledge of A and b, we must account for a possible relative error of order

$$O(\kappa(A)\varepsilon_{\text{mach}}).$$

If $\kappa(A) \approx 10^6$ and $\varepsilon_{\text{mach}} \approx 10^{-16}$, we do not expect more than about 10 accurate digits in the result.

To actually obtain these 10 accurate digits, one must also use an appropriate algorithm. In this course, we call an algorithm *stable* if it offers as much accuracy as the condition number of the problem permits—this is a rather poor definition of stability; the interested reader is directed to Lectures 14 and beyond in [TBI97] for more prudent definitions.

It turns out that the natural question "Is solving a linear system via LU factorization as described above a stable algorithm?" is particularly subtle. For practical purposes, the answer is yes, and we will leave it at that in this course. The interested reader is encouraged to look at [TBI97, Lecture 22].

3.3 Least squares problems

See Lecture 11 in [TBI97].

Let $(t_1, b_1), \ldots, (t_m, b_m)$ be given data points. We would like to compute a vector of coefficients $x = (a_d, \ldots, a_0)^T$ for a polynomial p of degree d defined by $p(t) = a_d t^d + \cdots + a_1 t + a_0$ such that $p(t_i) \approx b_i$. Specifically, we want to minimize the sum of squared errors:

$$\min_{x \in \mathbb{R}^{d+1}} \sum_{i=1}^{m} |p(t_i) - b_i|^2.$$

We assume $m > d + 1$, so that there are strictly more data than there are unknowns. Notice that

$$p(t_i) - b_i = \begin{bmatrix} t_i^d & t_i^{d-1} & \cdots & t_i^1 & t_i^0 \end{bmatrix} \begin{bmatrix} a_d \\ a_{d-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} - b_i.$$

Collecting all equations for $i = 1 \ldots m$ in a vector,

$$\begin{bmatrix} p(t_1) - b_1 \\ \vdots \\ \vdots \\ p(t_m) - b_m \end{bmatrix} = \underbrace{\begin{bmatrix} t_1^d & \cdots & t_1^0 \\ \vdots & & \vdots \\ \vdots & & \vdots \\ t_m^d & \cdots & t_m^0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_d \\ \vdots \\ a_0 \end{bmatrix}}_{x} - \underbrace{\begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_m \end{bmatrix}}_{b} = Ax - b.$$

Thus, the least squares problem reduces to:

Problem 3.8. *Given $A \in \mathbb{R}^{m \times n}$ of full column rank and $b \in \mathbb{R}^m$, compute $x \in \mathbb{R}^n$ solution of*

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|^2,$$

where $\|\cdot\|$ is the 2-norm: $\|y\|^2 = y_1^2 + \cdots + y_m^2$.

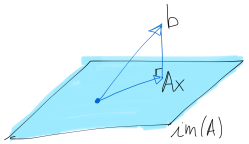
Above, $n = d + 1$. Matrix A as defined earlier indeed has full column rank provided all t_i's are distinct—we omit a proof.

Question 3.9. *Show that if A does not have full column rank, then the least squares problem cannot have a unique solution.*

■

If b is in the image of A (also called the range of A), that is, if b belongs to the subspace of linear combinations of the columns of A, then, by definition, there exists a solution x to the over-determined system of equations $Ax = b$. If that happens, x is a solution to the least squares problem (why?).

That is typically not the case. In general, for an over determined system, we de not expect b to be in the image of A. Instead, the solution x to the least squares problem is such that Ax is the vector in the image of A which is closest to b, in the 2-norm. In your introductory linear algebra course, you probably argued that this implies $b - Ax$ (the residue) is orthogonal to the image of A. Equivalently, $b - Ax$ is orthogonal to all columns of A (since they form a basis of the image of A). Algebraically:



$$A^T(b - Ax) = 0.$$

Reorganizing this statement yields the *normal equations* of $Ax = b$:

$$A^T Ax = A^T b.$$

Question 3.10. *Show that A^TA is invertible iff A has full column rank.*

■

As it turns out, solving the normal equations is a terrible way of solving a least-squares problem, and we can understand it by looking at the condition number of A^TA.

Problem 3.11. *In the 2-norm, show that $\kappa(A^TA) = \kappa(A)^2$, where we extend the definition $\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$ to rectangular matrices.*

■

Thus, if $\kappa(A) \approx 10^8$, the condition number of A^TA is already about 10^{16}, which means even if we solve the normal equations with a stable algorithm we cannot guarantee a single digit to be correct. Matrix A as defined above tends to be poorly conditioned if some of the t_i's are too close, so this is an issue for us.

Alternatively, if we have an algorithm to factor A into the product

$$A = \hat{Q}\hat{R},$$

where $\hat{Q} \in \mathbb{R}^{m\times n}$ has orthonormal columns and $\hat{R} \in \mathbb{R}^{n\times n}$ is upper triangular, then we can use that to solve the normal equations while avoiding the

squaring of the condition number. We proceed as follows. First, note that the following are equivalent:

$$A^T A x = A^T b$$
$$\hat{R}^T \hat{Q}^T \hat{Q} \hat{R} x = \hat{R}^T \hat{Q}^T b.$$

Question 3.12. *Show that $\hat{R}$ is invertible iff A has full column rank.*

■

Thus, using both invertibility of $\hat{R}^T$ (under our assumption that A has full column rank) and the fact that $\hat{Q}^T \hat{Q} = I$ (since $\hat{Q}$ has orthonormal columns), we find that the system simplifies to:

$$\hat{R} x = \hat{Q}^T b.$$

This is a triangular system. Importantly, its condition number is fine.

Question 3.13. *Show that in the 2-norm we have $\kappa(\hat{R}) = \kappa(A)$.*

■

The latter considerations motivate us to discuss algorithms to compute QR factorizations. You most likely already know one algorithm: the Gram–Schmidt process.

3.4 Conditioning of least squares problems

In the previous section, we swept two important points under the rug: (a) we did not formally argue that $\kappa(A)$, extended to rectangular matrices as $\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$, is a meaningful notion of conditioning; and (b) the right hand side of the normal equations is $A^T b$, which means that if b is perturbed to $b + \delta b$, then the right hand side of the normal equations is perturbed to $A^T b + A^T \delta b$ (plus round-off error). That is: the perturbation of the right hand side is not in just *any* direction; at least in exact arithmetic, it is necessarily in the image of A^T, which is orthogonal to the kernel of A. Both of these points mean the discussion about conditioning we had earlier, while useful to guide our intuition, is not entirely appropriate to discuss least squares problems. This section gives further details (it is not discussed in class, but is useful for homework.)

Lecture 18 in [TBI97] gives a proper treatment of conditioning for least squares. As a highlight of that lecture, we cover here one important question:

how sensitive is the least squares solution x to perturbations of b? Formally, let $\tilde{b} = b + \delta b$, where δb is some perturbation, and let $\tilde{x} = x + \delta x$ be the corresponding (perturbed) least squares solution. Assuming A has full column rank (as we always do in this chapter), the following equations hold:

$$x = (A^T A)^{-1} A^T b,$$
$$\tilde{x} = (A^T A)^{-1} A^T \tilde{b}.$$

Subtracting the first equation from the second, we infer that:

$$\delta x = (A^T A)^{-1} A^T \delta b.$$

The matrix $(A^T A)^{-1} A^T$ plays a special role: it is called the *pseudo-inverse* of A. We denote it by A^+:

$$A^+ = (A^T A)^{-1} A^T. \tag{3.3}$$

The pseudo-inverse gives the solutions to a least squares problem: $x = A^+ b$. Using the matrix norm subordinate to the vector 2-norm, we have that the norm of the perturbation on x is bounded as:

$$\|\delta x\| \leq \|A^+\| \|\delta b\|. \tag{3.4}$$

Thus, it is necessary to understand the norm of A^+, which is given by its largest singular value. To determine the singular values of A^+, first consider the SVD of A:

$$A = U \Sigma V^T, \qquad \text{where} \qquad \Sigma = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & \end{bmatrix} \in \mathbb{R}^{m \times n}$$

and $U \in \mathbb{R}^{m \times m}, V \in \mathbb{R}^{n \times n}$ are orthogonal matrices. A simple computation shows that:

$$A^+ = V \begin{bmatrix} 1/\sigma_1 & & \\ & \ddots & \\ & & 1/\sigma_n \\ & & \end{bmatrix} U^T.$$

Notice that this is an SVD of A^+ (up to the fact that one would normally order the singular values from largest to smallest, and they are here ordered from smallest to largest.) In particular, the largest singular value of A^+

is $1/\sigma_n$. This gives the following meaning to the definition we gave earlier without justification for the condition number of a full-rank, rectangular matrix:

$$\|A\| = \sigma_{\max}(A), \qquad \|A^+\| = \frac{1}{\sigma_{\min}(A)}, \qquad \kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \|A\|\|A^+\|.$$

(Compare this to the definition $\kappa(A) = \|A\|\|A^{-1}\|$ for square, invertible matrices.) We are now in a good position to reconsider (3.4):

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\|A^+\|\|\delta b\|}{\|x\|} = \underbrace{\kappa(A)\frac{\|b\|}{\|A\|\|x\|}}_{\substack{\text{relative sensitivity} \\ \text{of } x \text{ w.r.t. } b}} \frac{\|\delta b\|}{\|b\|}.$$

This inequality expresses the relative sensitivity of x with respect to perturbations of b in a least squares problem, in terms of the condition number of A and problem-dependent norms, $\|A\|, \|b\|, \|x\|$. We can further interpret this extra coefficient as follows:

$$\frac{\|b\|}{\|A\|\|x\|} = \frac{\|Ax\|}{\|A\|\|x\|}\frac{\|b\|}{\|Ax\|}.$$

The first factor is at most 1: it can only help (in that its effect on sensitivity can only be to lower it.) It indicates that it is preferable (from a sensitivity point of view) to have x in a subspace that is damped rather than amplified by A. The second factor has a good geometric interpretation. Consider the following:

Question 3.14. *Argue AA^+ is the orthogonal projector to the range of A.*

■

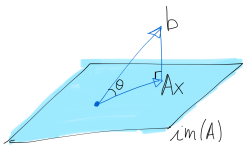
Based on the latter property of AA^+ and $x = A^+b$,

$$Ax = AA^+b$$

is the orthogonal projection of b to the range of A. Hence,

$$\frac{\|b\|}{\|Ax\|} = \frac{\|b\|}{\|AA^+b\|},$$

the ratio between the norm of b and that of its orthogonal projection to the range of A, is given by the cosine of the angle between b and that subspace.



Let θ denote this angle and let $\eta = \frac{\|A\|\|x\|}{\|Ax\|} \geq 1$. Then, we can summarize our findings as:

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\kappa(A)}{\eta \cos\theta} \frac{\|\delta b\|}{\|b\|} \leq \frac{\kappa(A)}{\cos\theta} \frac{\|\delta b\|}{\|b\|}.$$

Notice the role of θ: if b is close to the range of A, then $Ax \approx b$ "almost" has a consistent solution: this is rewarded by having $\cos\theta$ close to 1. On the other hand, if b is almost orthogonal to the range of A, then we are very far from having a consistent solution and sensitivity is exacerbated, as indicated by $\cos\theta$ close to 0.

3.5 Computing QR factorizations, $A = \hat{Q}\hat{R}$

See Lectures 7, 8 and 9 in [TBI97] in full, except for the part about continuous functions in Lecture 7 (we will discuss this in detail later in the course.) See also the Matlab codes on Blackboard (shown in class) for experiments showing the behavior of various QR algorithms.

Given a full column-rank matrix $A \in \mathbb{R}^{m\times n}$, classical Gram-Schmidt (CGS) computes a matrix $\hat{Q} \in \mathbb{R}^{m\times n}$ with orthonormal columns and an upper triangular matrix $\hat{R} \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that

$$A = \hat{Q}\hat{R}, \quad A = \begin{bmatrix} | & & | \\ a_1 & \cdots & a_n \\ | & & | \end{bmatrix}, \quad \hat{Q} = \begin{bmatrix} | & & | \\ q_1 & \cdots & q_n \\ | & & | \end{bmatrix}, \quad \hat{R} = \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}.$$

This can be expressed equivalently as a collection of n vector equations:

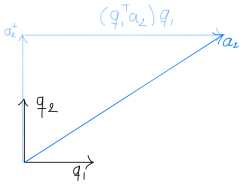
$$\begin{aligned} a_1 &= r_{11}q_1 \\ a_2 &= r_{12}q_1 + r_{22}q_2 \\ a_3 &= r_{13}q_1 + r_{23}q_2 + r_{33}q_3 \\ &\vdots \end{aligned} \tag{3.5}$$

This structure guarantees that

$$\text{span}(q_1) = \text{span}(a_1), \qquad \text{span}(q_1, q_2) = \text{span}(a_1, a_2), \text{ etc.}$$

To compute this factorization, CGS proceeds iteratively. First, it produces q_1 to span the same subspace as a_1, and to have unit norm:

$$q_1 = \frac{1}{\|a_1\|} a_1.$$



Then, it produces q_2 by projecting a_2 to the orthogonal complement of the space spanned by q_1,[3]

$$a_2^\perp = a_2 - (q_1^T a_2) q_1,$$

and by normalizing the result:[4]

$$q_2 = \frac{1}{\|a_2^\perp\|} a_2^\perp.$$

Similarly, q_3 is obtained by projecting a_3 to the orthogonal complement of the space spanned by q_1, q_2,

$$a_3^\perp = a_3 - (q_1^T a_3) q_1 - (q_2^T a_3) q_2,$$

and by normalizing the result:

$$q_3 = \frac{1}{\|a_3^\perp\|} a_3^\perp.$$

The procedure is continued until n vectors have been produced, with general formulas for j ranging from 1 to n:

$$a_j^\perp = a_j - \sum_{i=1}^{j-1} (q_i^T a_j) q_i, \qquad q_j = \frac{1}{\|a_j^\perp\|} a_j^\perp. \tag{3.6}$$

The above equations can be rewritten in the form of (3.5):

$$\begin{aligned}
a_1 &= \underbrace{\|a_1\|}_{r_{11}} q_1 \\
a_2 &= \underbrace{(q_1^T a_2)}_{r_{12}} q_1 + \underbrace{\|a_2^\perp\|}_{r_{22}} q_2 \\
a_3 &= \underbrace{(q_1^T a_3)}_{r_{13}} q_1 + \underbrace{(q_2^T a_3)}_{r_{23}} q_2 + \underbrace{\|a_3^\perp\|}_{r_{33}} q_3 \\
&\vdots
\end{aligned}$$

[3] This ensures $a_2^\perp$ is orthogonal to q_1, hence q_2 (which is just a scaled version of $a_2^\perp$) is also orthogonal to q_1, as desired. To get the formulas, we use the fact that these are equivalent: (a) project to the orthogonal complement of a space; and (b) project to the space, then subtract that from the original vector.

[4] We could also set $q_2 = -\frac{1}{\|a_2^\perp\|} a_2^\perp$: taking the positive sign is a convention; it will lead to a positive diagonal for R.

Algorithm 3.4 Classical Gram-Schmidt (unstable) [TBI97, Alg. 7.1]

Given: $A \in \mathbb{R}^{m\times n}$ with columns $a_1, \ldots, a_n \in \mathbb{R}^m$
for j in $1 \ldots n$ **do** ▷ For each column
  $v_j \leftarrow a_j$
  for i in $1 \ldots j-1$ **do** ▷ For each previously treated column
    $r_{ij} \leftarrow q_i^T a_j$ ▷ Compare with Algorithm 3.5
    $v_j \leftarrow v_j - r_{ij} q_i$
  end for
  $r_{jj} \leftarrow \|v_j\|$ ▷ By this point, $v_j = a_j^\perp$
  $q_j \leftarrow \frac{v_j}{r_{jj}}$
end for

The general formula for j ranging from 1 to n is:

$$a_j = \sum_{i=1}^{j-1} \underbrace{(q_i^T a_j)}_{r_{ij}} q_i + \underbrace{\|a_j^\perp\|}_{r_{jj}} q_j.$$

These show explicitly how to populate the matrices $\hat{Q}$ and $\hat{R}$ as we proceed through the Gram-Schmidt algorithm, see Algorithm 3.4. The following is equivalent Matlab code for this algorithm, organized a bit differently:

```
function [Q, R] = classical_gram_schmidt(A)
% Classical Gram-Schmidt orthonormalization algorithm (unstable)

    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1 : n

        v = A(:, j);

        R(1:j-1, j) = Q(:, 1:j-1)' * v;

        v = v - Q(:, 1:j-1) * R(1:j-1, j);

        R(j, j) = norm(v);
        Q(:, j) = v / R(j, j);

    end

end
```

See codes `lecture_LSQ_first_contact.m` and `lecture_QR_comparison.m` (shown in class).

Numerically, CGS is seen to behave rather poorly. What could possibly have gone wrong? Here is a list of suspects:

- Columns of $\hat{Q}$ are not really orthonormal?
- $\hat{R}$ is not really upper triangular?
- It is not true that $A = \hat{Q}\hat{R}$?
- A is not actually full column rank, breaking an important assumption?

Following the experiments in the codes on Blackboard, we find that A has full column rank, and when computing $A = \hat{Q}\hat{R}$ using CGS, we indeed satisfy $A \approx \hat{Q}\hat{R}$, and of course $\hat{R}$ is exactly upper triangular (by construction). The culprit: round-off errors cause columns of $\hat{Q}$ to be far from orthonormal in case A is poorly conditioned.

An improved CGS, called Modified Gram–Schmidt (MGS), partly fixes the problem—the modification is subtle; spend some time thinking about it. The main idea goes as follows.

Consider the master equation of CGS, namely (3.6). What does it mean for A to be poorly conditioned? It means its columns, while linearly *independent*, are *close* to being dependent, in the sense that a_j may be close to the subspace spanned by $a_1, \ldots, a_{j-1}$ or, equivalently, by $q_1, \ldots, q_{j-1}$. In turn, this means $a_j^\perp$ may be a (very) small vector: we have no control over that, it is a property of the given A. Thus, when forming $q_j = \frac{a_j^\perp}{\|a_j^\perp\|}$, any numerical errors accumulated in computing $a_j^\perp$ may be amplified by a lot. And indeed, we accumulate a lot of such errors, in 'random' directions, from the computation

$$a_j^\perp = a_j - (q_1^T a_j)q_1 - \cdots - (q_{j-1}^T a_j)q_{j-1}.$$

Each vector $(q_i^T a_j)q_i$ is rounded to some representable numbers, causing round-off errors, and these errors compound (add up) without any serious cancellations. Since these errors are added on top of the true $a_j^\perp$ which (in this discussion) is very small, the computed $a_j^\perp$ points in a direction that may be very far from being orthogonal to $q_1, \ldots, q_{j-1}$.

To alleviate this issue, the key is to notice that we can perform this computation differently, in such a way that errors introduced so far can be reduced going forward. As an example, consider

$$a_3^\perp = a_3 - (q_1^T a_3)q_1 - (q_2^T a_3)q_2.$$

In CGS, this formula is implemented as follows:

$$\begin{aligned} v_3 &\leftarrow a_3 \\ v_3 &\leftarrow v_3 - (q_1^T a_3) q_1 \\ v_3 &\leftarrow v_3 - (q_2^T a_3) q_2. \end{aligned}$$

That is: initialize v_3 to a_3; project a_3 to span(q_1), and subtract that from v_3; then project a_3 to q_2 and subtract that from v_3. Mathematically, $v_3 = a_3^\perp$ in the end. But we could do this computation differently. Indeed, assume we only executed the first step, so that, at this point, $v_3 = a_3$. Then, surely, $q_1^T v_3 = q_1^T a_3$. Now assume we performed the first two steps, so that at this point

$$v_3 = a_3 - (q_1^T a_3) q_1.$$

Then, mathematically,

$$q_2^T v_3 = q_2^T a_3 - (q_1^T a_3)(q_2^T q_1) = q_2^T a_3,$$

since $q_1^T q_2 = 0$. Thus, the following procedure also computes $a_3^\perp$:

$$\begin{aligned} v_3 &\leftarrow a_3 \\ v_3 &\leftarrow v_3 - (q_1^T v_3) q_1 = (I - q_1 q_1^T) v_3 \\ v_3 &\leftarrow v_3 - (q_2^T v_3) q_2 = (I - q_2 q_2^T) v_3. \end{aligned}$$

This procedure reads as follows: initialize v_3 to a_3; project v_3 to the orthogonal complement of q_1; and project the result to the orthogonal complement of q_2. More generally, the rule to obtain $a_j^\perp$ is:

$$a_j^\perp = (I - q_{j-1} q_{j-1}^T) \cdots (I - q_1 q_1^T) a_j.$$

This small modification turns out to be beneficial. Why? Follow the numerical errors: each projection $I - q_i q_i^T$ introduces some round-off error. Importantly, much of that round-off error will be eliminated by the next projector, $I - q_{i+1} q_{i+1}^T$, because components of the error that happen to be aligned with q_{i+1} will be mathematically zeroed out (numerically, they are greatly reduced). One concrete effect for example is that q_j is as orthogonal to q_{j-1} as numerically possibly, because the last step in obtaining $a_j^\perp$ is to apply the projector $I - q_{j-1} q_{j-1}^T$. This is not so for CGS.

This modification of CGS, called MGS, is presented in a neatly organized procedure as Algorithm 3.5. The algorithm is organized somewhat differently from Algorithm 3.4 (specifically, it applies the projector $I - q_i q_i^T$ to all subsequent v_j's as soon as q_i becomes available), but other than that it is equivalent to taking Algorithm 3.4 and only replacing $r_{ij} \leftarrow q_i^T a_j$ with $r_{ij} \leftarrow q_i^T v_j$. Here is Matlab code for MGS.

Algorithm 3.5 Modified Gram-Schmidt [TBI97, Alg. 8.1]

Given: $A \in \mathbb{R}^{m \times n}$ with columns $a_1, \ldots, a_n \in \mathbb{R}^m$
for i in $1 \ldots n$ **do**
    $v_i \leftarrow a_i$
end for
for i in $1 \ldots n$ **do**
    $r_{ii} \leftarrow \|v_i\|$
    $q_i \leftarrow \frac{v_i}{r_{ii}}$
    for j in $(i+1) \ldots n$ **do**
        $r_{ij} \leftarrow q_i^T v_j$ ▷ Crucially, here we compute $q_i^T v_j$, not $q_i^T a_j$
        $v_j \leftarrow v_j - r_{ij} q_i$
    end for
end for

```
function [Q, R] = modified_gram_schmidt(A)
% Modified Gram-Schmidt orthonormalization algorithm (better ...
    stability than CGS)

    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1 : n

        v = A(:, j);

        R(j, j) = norm(v);
        Q(:, j) = v / R(j, j);

        R(j, (j+1):n) = Q(:, j)' * A(:, (j+1):n);
        A(:, (j+1):n) = A(:, (j+1):n) - Q(:, j) * R(j, (j+1):n);

    end

end
```

MGS provides much better stability than CGS, but it is still not perfect. In particular, while it is still true that $A \approx \hat{Q}\hat{R}$, and indeed $\hat{Q}$ is closer to having orthonormal columns than when we use CGS, it may still be far from orthonormality when A is poorly conditioned. We discuss fixes below. Before we move on, ponder the following questions:

1. What is the computational cost of MGS?

2. What are the failure modes (assuming exact arithmetic)?

3. Given a matrix $A \in \mathbb{R}^{m\times n}$, do we always have existence of a factorization $A = \hat{Q}\hat{R}$? Do we have uniqueness?

All the answers (with developments) are in [TBI97, Lec. 7–9]. In a nutshell:

1. The cost is dominated by the nested loop (instructions 9 and 10):

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 4m-1 = (4m-1)\frac{n(n-1)}{2} = \sim 2mn^2 \text{ flops;}$$

2. Failure can only occur if we divide by 0, meaning $r_{jj} = 0$ for some j. This happens if and only if $a_j^{\perp} = 0$, that is, iff a_j is in the span of $a_1, \ldots, a_{j-1}$, meaning the columns of A are not linearly independent. Hence, mathematically, if A has full column rank, then MGS (and CGS) succeed (in exact arithmetic).

3. If A has full column rank, we have existence of course (since the algorithm produces such a factorization). As for uniqueness, the only freedom we have (as a result of $\hat{R}$ being upper triangular) is in the sign of r_{jj}. By forcing $r_{jj} > 0$, the factorization is unique.

As a side note: if A is not full column rank, there still exists a $\hat{Q}\hat{R}$ factorization, but it is not unique and its properties are different from the above: see [TBI97, Thm. 7.1]. Furthermore, the reason we write the factorization as $A = \hat{Q}\hat{R}$ instead of simply $A = QR$ is because the latter notation is traditionally reserved for the "full" QR decomposition (as opposed to the "economy size" or "thin" or "partial" QR we have discussed here), in which $Q \in \mathbb{R}^{m\times m}$ is orthogonal (square matrix) and $R \in \mathbb{R}^{m\times n}$ has an $n \times n$ upper triangular block at the top, followed by rows of zeros. Matlab's `qr(A)` produces the full QR decomposition by default. To get the economy size, call `qr(A, 0)`.

3.6 Least-squares via SVD

As conditioning of A is pushed to terrible values, all algorithms will break. When facing particularly hard problems, it may pay to use the following algorithm for least squares:

1. Compute the thin SVD of A: $A = \hat{U}\hat{\Sigma}\hat{V}^T$, where $\hat{U} \in \mathbb{R}^{m\times n}, \hat{V} \in \mathbb{R}^{n\times n}$ have orthonormal columns and $\hat{\Sigma} = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$.
   Matlab: `[U, S, V] = svd(A, 0);`

2. Plug this into the normal equations and cancel terms using orthonormality of $\hat{U}, \hat{V}$ and invertibility of $\hat{\Sigma}$:

$$\begin{aligned} A^T A x &= A^T b \\ \hat{V}\hat{\Sigma}^T\hat{U}^T\hat{U}\hat{\Sigma}\hat{V}^T x &= \hat{V}\hat{\Sigma}^T\hat{U}^T b \\ \hat{\Sigma}\hat{V}^T x &= \hat{U}^T b \\ x &= \hat{V}\hat{\Sigma}^{-1}\hat{U}^T b, \end{aligned}$$

where $\hat{\Sigma}^{-1} = \mathrm{diag}(1/\sigma_1, \ldots, 1/\sigma_n)$ is trivial to apply.

This algorithm is the most stable of all the ones we discuss in these notes. The drawback is that it requires computing the SVD of A, which is typically more expensive than computing a QR factorization: more on this later.

3.7 Regularization

Poor conditioning is often fought with regularization (on top of using stable algorithms as above). We do not discuss this much in this course; keywords are: *Tikhonov regularization*, or *regularized least squares*.

The general idea is to minimize $\|Ax - b\|^2 + \lambda\|x\|^2$, with some chosen $\lambda > 0$, instead of merely minimizing $\|Ax - b\|^2$. This intuitively promotes smaller solutions x by penalizing (we say, *regularizing*) the norm of x. In practice, this is done as follows: given a least squares problem $Ax = b$, replace it with the larger least squares problem $\tilde{A}x = \tilde{b}$:

$$\underbrace{\begin{bmatrix} A \\ \sqrt{\lambda} I_n \end{bmatrix}}_{\tilde{A}} x = \underbrace{\begin{bmatrix} b \\ 0 \end{bmatrix}}_{\tilde{b}}. \tag{3.7}$$

As announced, it corresponds to minimizing $\|\tilde{A}x - \tilde{b}\|^2 = \|Ax - b\|^2 + \lambda\|x\|^2$. The normal equations involve $\tilde{A}^T\tilde{A} = A^T A + \lambda I$, and the right hand side is $\tilde{A}^T\tilde{b} = A^T b$. Problem (3.7) is solved by applying any of the good methods described above on $\tilde{A}, \tilde{b}$ directly. For example, you can apply MGS (suitably "fixed" as described below if necessary) to compute a QR factorization of $\tilde{A}$.

What is the condition number of this new problem? It is easy to see that the eigenvalues λ_i of $\tilde{A}^T\tilde{A} = A^T A + \lambda I$ are simply the squared singular values of A shifted by λ. Thus, the singular values of $\tilde{A}$ are:

$$\sigma_i(\tilde{A}) = \sqrt{\lambda_i(\tilde{A}^T\tilde{A})} = \sqrt{\sigma_i(A)^2 + \lambda^2}.$$

Hence, the condition number of $\tilde{A}$ is

$$\kappa(\tilde{A}) = \sqrt{\frac{\sigma_{\max}(A)^2 + \lambda^2}{\sigma_{\min}(A)^2 + \lambda^2}}.$$

Clearly, for $\lambda = 0$, $\kappa(\tilde{A}) = \kappa(A)$, while for larger values of λ, the condition number of $\tilde{A}$ decreases, thus reducing numerical difficulties. On the other hand, larger values of λ change the problem (and its solution): which value of λ is appropriate depends on the application.

3.8 Fixing MGS: twice is enough

MGS works significantly better than CGS, but it can still break for poorly conditioned A. One easy way to improve MGS is known as the *twice is enough* trick. The goal is to have $\hat{Q}$ indeed very close to having orthonormal columns, while preserving $A \approx \hat{Q}\hat{R}$ (which MGS already delivers in practice). It works as follows.

If A has full column rank but it is ill-conditioned, then computing

$$A \xrightarrow{\text{MGS}} \hat{Q}_1\hat{R}_1$$

using MGS results in $A \approx \hat{Q}_1\hat{R}_1$ yet columns of $\hat{Q}_1$ are not quite orthonormal. Importantly though, $\hat{Q}_1$ is much better conditioned than A: after all, even though it is not quite orthogonal, it is much closer to being orthogonal than A itself, and orthogonal matrices have condition number 1 in the 2-norm. The trick is to apply MGS a second time, this time to $\hat{Q}_1$:

$$\hat{Q}_1 \xrightarrow{\text{MGS}} \hat{Q}_2\hat{R}_2.$$

This time, we expect columns of $\hat{Q}_2$ to be very nearly orthonormal. Furthermore, since $\hat{Q}_1 \approx \hat{Q}_2\hat{R}_2$,

$$A \approx \hat{Q}_1\hat{R}_1 \approx \hat{Q}_2(\hat{R}_2\hat{R}_1).$$

Defining $\hat{Q} = \hat{Q}_2$ and $\hat{R} = \hat{R}_2\hat{R}_1$ (which is indeed upper triangular because it is a product of two upper triangular matrices) typically results in an excellent QR factorization of A. In the experiments on Blackboard, this QR is as good as the one built into Matlab as the function `qr`. (The latter is based on Householder triangularization.) It is however slower, since Householder triangularization is cheaper than two calls to MGS (by some constant factor.)

3.9 Solving least-squares with MGS directly

Another "fix" is described in Lecture 19 of [TBI97]. It consists not in fixing the QR factorization itself, but rather in using it differently for the purpose of solving a least squares problem. Specifically, go through the reasoning of plugging $A = \hat{Q}\hat{R}$ in the normal equations again, but not using orthonormality:

$$\begin{aligned} A^T A x &= A^T b \\ \hat{R}^T \hat{Q}^T \hat{Q} \hat{R} x &= \hat{R}^T \hat{Q}^T b \\ \hat{Q}^T \hat{Q} \hat{R} x &= \hat{Q}^T b. \end{aligned} \tag{3.8}$$

At this point, instead of canceling $\hat{Q}^T\hat{Q}$, we *could* solve these systems:

$$\begin{aligned} \hat{Q}^T \hat{Q} y &= \hat{Q}^T b \\ \hat{R} x &= y. \end{aligned}$$

We would expect this to fare better, because $\hat{Q}^T\hat{Q}$ ought to be better conditioned than $A^T A$. However, this would be fairly expensive, as the system $\hat{Q}^T\hat{Q}$ does not have particularly favorable structure (such as triangularity).

Instead, we do the following: augment the matrix A with the vector b, and compute a QR factorization of that matrix (for example, using simple MGS):

$$\tilde{A} = \begin{bmatrix} A & b \end{bmatrix} \xrightarrow{\text{MGS}} \tilde{Q}\tilde{R}.$$

Gram–Schmidt proceeds column by column: for the first n of these, the fact we appended b to the matrix has no effect, so we get the exact same factorization at first. This is why $\hat{Q}$ and $\hat{R}$ appear in $\tilde{Q}$ and $\tilde{R}$.

Consider also the QR factorization $A = \hat{Q}\hat{R}$. Since this factorization is unique, it must be that the QR factorization of $\tilde{A}$ admits the following form:

$$\tilde{A} = \begin{bmatrix} A & b \end{bmatrix} = \underbrace{\begin{bmatrix} \hat{Q} & q \end{bmatrix}}_{\tilde{Q}} \underbrace{\begin{bmatrix} \hat{R} & r \\ 0 & \alpha \end{bmatrix}}_{\tilde{R}} = \begin{bmatrix} \hat{Q}\hat{R} & \hat{Q}r + \alpha q \end{bmatrix},$$

where $q \in \mathbb{R}^m$ and $\alpha \in \mathbb{R}$. Consider the last column of this matrix equality:

$$b = \hat{Q}r + \alpha q.$$

Mathematically, $\hat{Q}^T q = 0$. Numerically, this might not be quite true, but, intuitively, making that assumption should be safer than making the strong (and false) assumption $\hat{Q}^T\hat{Q} \approx I$. Thus, we expect the following mathematical identity to hold approximately numerically:

$$\hat{Q}^T b = \hat{Q}^T \hat{Q} r.$$

Plug this into (3.8):

$$\hat{Q}^T\hat{Q}\hat{R}x = \hat{Q}^T b = \hat{Q}^T\hat{Q}r.$$

Simply using that $\hat{Q}^T\hat{Q}$ is expected to be invertible (a far safer assumption than orthonormality), we find:

$$\hat{R}x = r.$$

In summary: compute the QR factorization $\tilde{Q}\tilde{R}$ of $\begin{bmatrix} A & b \end{bmatrix}$, extract the top-left block of $\tilde{R}$, namely, $\hat{R}$, and extract the top-right column of $\tilde{R}$, namely, r; then solve the triangular system $\hat{R}x = r$. This turns out to yield a numerically well-behaved solution to the least-squares problem.

Chapter 4

Solving simultaneous nonlinear equations

In this chapter, we are interested in solving the following problem.

Problem 4.1 (Simultaneous nonlinear equations)**.** *Given $f\colon \mathbb{R}^n \to \mathbb{R}^n$, continuous on a non-empty, closed domain $D \subseteq \mathbb{R}^n$, find $\xi \in D$ such that $f(\xi) = 0$.*

This is a generalization of the very first problem we encountered in this course. Notice that solving systems of linear equations as we did in the previous chapter is also a special case, by considering $f(x) = Ax - b$. These notes follow the narrative in class—see Chapter 4 in [SM03] for the full story. Ponder for a moment why we require f to be continuous.

For example, consider solving $f(x) = 0$ with $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ defined as:

$$f(x) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix},$$

where

$$\begin{aligned} f_1(x_1, x_2) &= \frac{1}{9}\left(x_1^2 + x_2^2 - 9\right), \\ f_2(x_1, x_2) &= -x_1 + x_2 - 1. \end{aligned}$$

The locus $f_1 = 0$, that is, the set of points $x \in \mathbb{R}^2$ such that $f_1(x) = 0$ is the circle of radius 3 centered at the origin. The locus $f_2 = 0$ is the line passing through the points $(0, 1)$ and $(-1, 0)$. The two loci intersect at two points: these are the solutions to the system $f(x) = 0$. See Figure 4.1

We could solve this particular system analytically, but in general this task may prove difficult. Thus, we set out to study general algorithms to compute

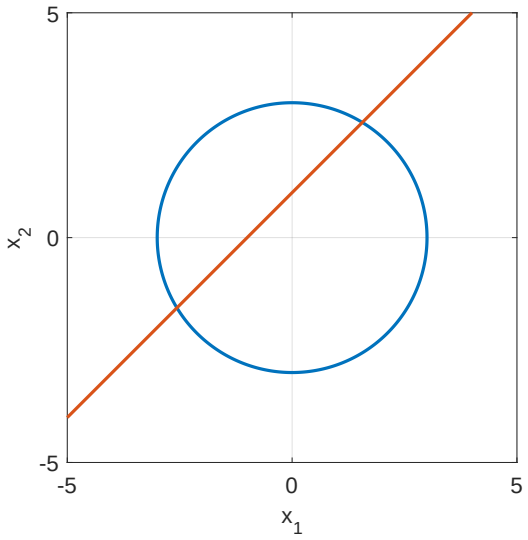


Figure 4.1: Loci $f_1(x) = 0$ and $f_2(x) = 0$. Intersections solve $f(x) = 0$.

roots of f in n dimensions. Looking back at our work for single nonlinear equations, consider the algorithms we covered then and ponder: which of these have a good chance to generalize? Which seem difficult to generalize?

4.1 Simultaneous iteration

As for simple iteration, the core idea is: given f as above, pick $g\colon \mathbb{R}^n \to \mathbb{R}^n$ with the property that $f(\xi) = 0 \iff g(\xi) = \xi$. Then, root finding becomes equivalent to computing fixed points. If furthermore $g(D) \subseteq D$ (which means: for any $x \in D$, $g(x) \in D$), given $x^{(0)} \in D$ we can setup a sequence:

$$x^{(k+1)} = g(x^{(k)}).$$

Notice we used superscripts to index elements of a sequence now, so that subscripts still index entries of a vector, as usual. Hence, $x_i^{(k)}$ is the ith entry of the kth element of the sequence.

One way to create functions g in one dimension was *relaxation*, which considers

$$g(x) = x - \lambda f(x)$$

for some nonzero $\lambda \in \mathbb{R}$. Let's see how this fares on our example.

```
%% Plot the loci
t = linspace(0, 2*pi, 2501);
plot(3*cos(t), 3*sin(t), 'LineWidth', 2);
hold all;
t = linspace(-5, 5, 251);
plot(t, 1+t, 'LineWidth', 2);
axis equal;
xlabel('x_1'); ylabel('x_2');

xlim([-5, 5]);
ylim([-5, 5]);

%% Run simultaneous iteration
lambda = 0.05; % try different values here; what happens?
f = @(x) [ (x(1)^2 + x(2)^2 - 9)/9 ; -x(1) + x(2) - 1 ];
g = @(x) x - lambda*f(x);
x = [1 ; 0];

plot(x(1), x(2), 'k.', 'MarkerSize', 15);
fprintf('%.8g   %.8g \n', x(1), x(2));

for k = 1 : 100

    x = g(x);

    plot(x(1), x(2), 'k.', 'MarkerSize', 15);
    fprintf('%0.8g   %0.8g \n', x(1), x(2));

end

f(x)
```

This creates the following (truncated) output, with Figure 4.2.

```
1           0
1.0444444   0.1
1.0883285   0.19722222
1.1315321   0.29177754
1.173946    0.38376527
1.2154714   0.4732743
1.2560194   0.56038416
1.2955105   0.64516592
1.3338739   0.72768315
1.3710475   0.80799268
1.4069774   0.88614543
1.4416172   0.96218702
```

```
1.4749279   1.0361585
% ...
1.6138751   2.6717119
1.6097494   2.66882
1.6057833   2.6658665
1.6019756   2.6628623
1.5983247   2.659818
1.5948288   2.6567433
1.5914856   2.6536476
1.588293    2.6505395
1.5852484   2.6474272
1.582349    2.6443182
1.5795921   2.6412198
1.5769746   2.6381384
1.5744933   2.6350802
1.5721451   2.6320509

ans =  % this is f(x) at the last iterate

0.0444
0.0599
```

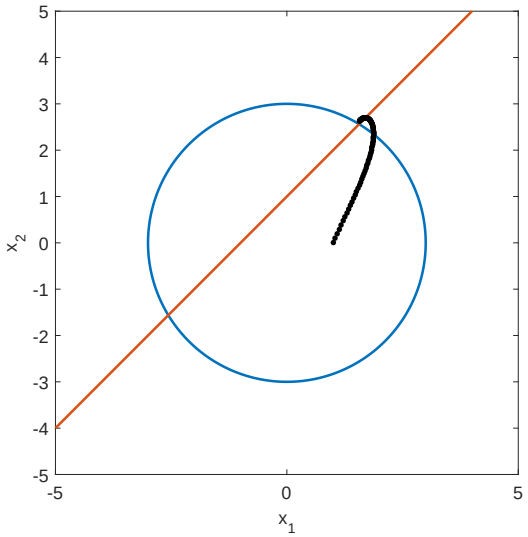


Figure 4.2: Relaxation with $\lambda = 0.05$, $x^{(0)} = (1, 0)^T$.

It seems to converge to the root in the positive orthant. Play around

with the parameters. You should find that it easily diverges. I never saw it converge to the other root.

Thus, there is at least *hope* for simultaneous iteration. To understand convergence, we turn toward the contraction mapping theorem we had in one dimension, and try to generalize it to $\mathbb{R}^n$.

4.2 Contractions in $\mathbb{R}^n$

We shall prove Theorem 4.1 in [SM03].

Theorem 4.2 (Contraction mapping theorem in $\mathbb{R}^n$)**.** *Suppose D is a nonempty, closed subset of $\mathbb{R}^n$, $g\colon \mathbb{R}^n \to \mathbb{R}^n$ is continuous on D, and $g(D) \subseteq D$. Suppose further that g is a* contraction *on D (in some norm). Then, g has a unique fixed point ξ in D and simultaneous iteration converges to ξ for any $x^{(0)}$ in D.*

To make sense of this, we need to introduce a notion of contraction in $\mathbb{R}^n$. This is Definition 4.2 in [SM03].

Definition 4.3. *Suppose $g\colon \mathbb{R}^n \to \mathbb{R}^n$ is defined on a closed subset D of $\mathbb{R}^n$. If there exists L such that*

$$\|g(x) - g(y)\| \leq L\|x - y\|$$

for all $x, y \in D$ for some vector norm $\|\cdot\|$, we say g satisfies a Lipschitz condition *on D. L is the* Lipschitz constant *in that norm. In particular, if $L \in (0, 1)$ and $g(D) \subseteq D$, g is a* contraction *on D in that norm.*

Here are some typical vector norms that are often useful, called the 2-norm, 1-norm and ∞-norm respectively:

- $\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$,
- $\|x\|_1 = |x_1| + \cdots + |x_n|$,
- $\|x\|_\infty = \max_{i \in \{1,\ldots,n\}} |x_i|$.

See [TBI97, Lec. 3] for further details, including the associated subordinate matrix norms: you should be comfortable with these notions. A couple of remarks are in order.

1. If g satisfies a Lipschitz condition, then g is continuous on D. Often, we say g is *Lipschitz continuous*.

2. All norms in $\mathbb{R}^n$ are equivalent, that is, if $\|\cdot\|_a, \|\cdot\|_b$ are any two vector norms, there exist constants $c, C > 0$ such that for any vector x, we have

$$c\|x\|_b \leq \|x\|_a \leq C\|x\|_b.$$

   Here is one consequence: if $\|x^{(k)} - \xi\| \to 0$ in some norm, then the same is true in any other norm.

3. If g satisfies a Lipschitz condition on D in any norm, then it satisfies it in all norms, possibly with another constant—can you prove it? Importantly, g may be a contraction in some norm, yet *not* be a contraction in another norm (and vice versa).

Proof of Theorem 4.2. The theorem has three claims:

1. Existence of a fixed point ξ;
2. Uniqueness of the fixed point ξ;
3. Convergence of simultaneous iteration to ξ.

The key point here is to notice that in the proofs of *uniqueness* and *convergence*, we *assume* existence of a fixed point. Thus, these proofs do not achieve much on their own: we need to prove *existence* separately.

If we assume existence for now, then we can easily show parts 2 and 3. The technical part is proving existence, which we will do later.

Uniqueness. Assume there exists a fixed point $\xi \in D$, that is, $g(\xi) = \xi$. For contradiction, assume there exists another fixed point $\eta \in D$. Then,

$$\|\xi - \eta\| = \|g(\xi) - g(\eta)\| \leq L\|\xi - \eta\|.$$

Since $\xi \neq \eta$ by assumption, we can divide by $\|\xi - \eta\|$ and get $L \geq 1$, which contradicts the fact g is a contraction.

Convergence. Still assume there exists a fixed point ξ. Then,

$$\|x^{(k+1)} - \xi\| = \|g(x^{(k)}) - g(\xi)\| \leq L\|x^{(k)} - \xi\| \leq L^{k+1}\|x^{(0)} - \xi\|.$$

Since $L \in (0, 1)$, we deduce

$$\lim_{k\to\infty} \|x^{(k)} - \xi\| \leq \lim_{k\to\infty} L^k\|x^{(0)} - \xi\| = 0,$$

so that $x^{(k)} \to \xi$. We even showed the error $\|x^{(k)} - \xi\|$ converges to 0 at least linearly. □

To prove existence of a fixed point ξ, we use the notion of Cauchy sequence.

For a single equation, we proved existence using the intermediate value theorem: that does not generalize to higher dimensions, which is why we need the extra work.

Definition 4.4. *A sequence $x^{(0)}, x^{(1)}, x^{(2)}, \dots$ in $\mathbb{R}^n$ is a* Cauchy sequence *in $\mathbb{R}^n$ if for any $\varepsilon > 0$ there exists k_0 (which usually depends on ε) such that:*

$$\forall k, m \geq k_0, \qquad \|x^{(m)} - x^{(k)}\| \leq \varepsilon.$$

This is independent of the choice of norm, since norms are equivalent in $\mathbb{R}^n$.

Here are a few facts related to this notion, which we shall not prove.

1. $\mathbb{R}^n$ is *complete*, that is: every Cauchy sequence in $\mathbb{R}^n$ converges in $\mathbb{R}^n$.
2. If all iterates of a Cauchy sequence are in a closed set $D \subseteq \mathbb{R}^n$, then the limit ξ exists and is in D—see [SM03, Lemma 4.1].
3. If g is continuous on D and the sequence $x^{(0)}, x^{(1)}, \dots$ in D converges to ξ in D, then $\lim_{k\to\infty} g(x^{(k)}) = g\left(\lim_{k\to\infty} x^{(k)}\right) = g(\xi)$—this is a direct consequence of continuity of f, see [SM03, Lemma 4.2].

We can now proceed with the proof.

Proof of Theorem 4.2, continued. **Existence.** All that is left to do to prove existence of a fixed point is to show the simultaneous iteration sequence is Cauchy. Indeed, if it is, then the second point above asserts $\lim_{k\to\infty} x^{(k)} = \xi \in D$, and the third point yields

$$\xi = \lim_{k\to\infty} x^{(k+1)} = \lim_{k\to\infty} g(x^{(k)}) = g\left(\lim_{k\to\infty} x^{(k)}\right) = g(\xi),$$

so that ξ is a fixed point in D. To establish that the sequence is Cauchy, we must control the distance between any two iterates. Without loss of generality, let $m > k$. By the triangle inequality:

$$\begin{aligned}\|x^{(m)} - x^{(k)}\| &= \|x^{(m)} - x^{(m-1)} + x^{(m-1)} - x^{(m-2)} + \cdots + x^{(k+1)} - x^{(k)}\| \\ &\leq \|x^{(m)} - x^{(m-1)}\| + \|x^{(m-1)} - x^{(m-2)}\| + \cdots + \|x^{(k+1)} - x^{(k)}\|.\end{aligned}$$

All of these terms are of the same form. Let's look at one of them:

$$\|x^{(k+1)} - x^{(k)}\| = \|g(x^{(k)}) - g(x^{(k-1)})\| \leq L\|x^{(k)} - x^{(k-1)}\|.$$

By induction,

$$\|x^{(k+1)} - x^{(k)}\| \leq L^k\|x^{(1)} - x^{(0)}\|.$$

Applying this to each term in the sum above, we get

$$\begin{aligned}\|x^{(m)} - x^{(k)}\| &\leq \left(L^{m-1} + L^{m-2} + \cdots + L^k\right) \|x^{(1)} - x^{(0)}\| \\ &= \left(L^{m-1-k} + L^{m-2-k} + \cdots + 1\right) L^k \|x^{(1)} - x^{(0)}\|.\end{aligned}$$

Let $S = 1 + L + L^2 + \cdots$. Notice that $LS = S - 1$. Hence, $S = \frac{1}{1-L}$.

The geometric sum is easily understood:

$$1 + L + L^2 + \cdots + L^{m-2-k} + L^{m-1-k} \leq 1 + L + L^2 + \cdots = \frac{1}{1-L}.$$

Hence,

$$\|x^{(m)} - x^{(k)}\| \leq \frac{\|x^{(1)} - x^{(0)}\|}{1-L} L^k.$$

The fraction is just some number. Since $L \in (0, 1)$, for any ε, we can pick k_0 such that the right hand side is less than ε for all $k \geq k_0$ (and $m > k$ as assumed earlier); thus, the sequence is Cauchy. □

4.3 Jacobians and convergence

For a single nonlinear equation, we showed that if $g\colon \mathbb{R} \to \mathbb{R}$ is continuously differentiable, then the asymptotic behavior of simple iteration near a fixed point ξ ultimately depends on $|g'(\xi)|$ alone. In particular, if $|g'(\xi)| < 1$, then g is locally a contraction around ξ, and $|g'(\xi)|$ dictates the asymptotic rate of convergence.

Here, we want to develop analogous understanding of contractions in $\mathbb{R}^n$. The analog of $g'(\xi)$ will be the Jacobian of g at ξ (a matrix), and the analog of the magnitude (absolute value) will be to take a (subordinate) matrix norm. First, a multivariable calculus reminder [SM03, Def. 4.3]

Definition 4.5. *Let $g\colon \mathbb{R}^n \to \mathbb{R}^n$ be continuous on an open neighborhood*[1] *$N(x)$ of $x \in \mathbb{R}^n$. Suppose all first partial derivatives $\frac{\partial g_i}{\partial x_j}$ exist at x. The* Jacobian matrix *$J_g(x)$ of g at x is the $n \times n$ matrix with elements*

$$(J_g(x))_{ij} = \frac{\partial g_i}{\partial x_j}(x), \qquad 1 \leq i, j \leq n.$$

A few remarks.

1. $J_g\colon \mathbb{R}^n \to \mathbb{R}^{n \times n}$ associates a matrix to each vector x (where it is defined).

[1] An open neighborhood of x in some set D is any open subset of D which contains x.

2. If g is differentiable at x (for example, if the partial derivatives are continuous in $N(x)$), then $J_g(x)$ represents the differential of g at x, that is:

$$g(x+h) = g(x) + J_g(x)h + E(x,h),$$

where $E(x,h)$ "goes to zero faster than h when h goes to zero"; that is:

$$\lim_{h\to 0} \frac{\|E(x,h)\|}{\|h\|} = 0,$$

for any vector norm $\|\cdot\|$.

3. What this limit means is: for *any* tolerance $\varepsilon > 0$ (of our own choosing), there exists a bound $\delta > 0$ (perhaps very small) such that the fraction is smaller than ε provided $\|h\| \le \delta$. Explicitly:

$$\forall \varepsilon > 0,\ \exists \delta > 0 \text{ such that if } \|h\| \le \delta,\ \text{then } \|E(x,h)\| \le \varepsilon \|h\|.$$

4. If g is *twice* differentiable at x, then by Taylor we have $\|E(x,h)\| \le c\|h\|^2$ for some constant c, provided h is sufficiently small. We write:

$$g(x+h) = g(x) + J_g(x)h + O(\|h\|^2). \tag{4.1}$$

Recalling the concept of subordinate matrix norm, we can understand how the Jacobian at a fixed point ξ dictates local behavior of simultaneous iteration. Assume g is differentiable at ξ. Let $x^{(k)} = \xi + h$ and think of h as small enough so that $\|E(\xi,h)\| \le \varepsilon\|h\|$—we will pick ε momentarily. Then,

$$\begin{aligned}
\|x^{(k+1)} - \xi\| = \|g(x^{(k)}) - g(\xi)\| &= \|g(\xi+h) - g(\xi)\| \\
&= \|g(\xi) + J_g(\xi)h + E(\xi,h) - g(\xi)\| \\
&= \|J_g(\xi)h + E(\xi,h)\| \\
\text{(triangle inequality)} \quad &\le \|J_g(\xi)h\| + \|E(\xi,h)\| \\
\text{(subordinate norm)} \quad &\le \|J_g(\xi)\|\|h\| + \varepsilon\|h\| \\
&= \big(\|J_g(\xi)\| + \varepsilon\big)\|x^{(k)} - \xi\|.
\end{aligned}$$

We can ensure $x^{(k+1)}$ is closer to ξ than $x^{(k)}$ is close to ξ in the chosen norm if we can make $\|J_g(\xi)\| + \varepsilon < 1$. Surely, if $\|J_g(\xi)\| \ge 1$, then we lose. But, as soon as $\|J_g(\xi)\| < 1$, there exists $\varepsilon > 0$ small enough such that $\|J_g(\xi)\| + \varepsilon$ is still strictly less than 1. And since we get to pick ε, we can do that. The only downside is that it may require us to force $\|h\|$ to be very small, that is: this may only be useful if $x^{(k)}$ is very close to ξ. Here is the take-away:

If ξ is a fixed point of g and g is differentiable at ξ, and if $\|J_g(\xi)\| < 1$ in some subordinate matrix norm, then there exists a (possibly very small) neighborhood of ξ such that, if $x^{(0)}$ is in that neighborhood, then simultaneous iteration converges to ξ. Furthermore, the errors $\|x^{(k)} - \xi\|$ converge to zero at least linearly. By extension, in that scenario, we say that the sequence $x^{(0)}, x^{(1)}, \ldots$ converges to ξ at least linearly.

The discussion above essentially captures [SM03, Theorem 4.2].

An example to practice the calculus. Consider f as defined above and the associated relaxation, as well as their Jacobians:

$$f(x) = \begin{bmatrix} \frac{1}{9}(x_1^2 + x_2^2 - 9) \\ -x_1 + x_2 - 1 \end{bmatrix}, \quad g(x) = x - \lambda f(x)$$

$$J_f(x) = \begin{bmatrix} \frac{2}{9}x_1 & \frac{2}{9}x_2 \\ -1 & 1 \end{bmatrix}, \qquad J_g(x) = I_2 - \lambda J_f(x) = \begin{bmatrix} 1 - \frac{2\lambda}{9}x_1 & -\frac{2\lambda}{9}x_2 \\ \lambda & 1 - \lambda \end{bmatrix}.$$

What is the norm of J_g at ξ in the positive orthant? We investigate this numerically for a range of λ values, showing the results in Figure 4.3.

```
% Define the problem
f  = @(x) [ (x(1)^2 + x(2)^2 - 9)/9 ; -x(1) + x(2) - 1 ];
Jf = @(x) [ 2*x(1)/9, 2*x(2)/9 ; -1, 1 ];

% Get a root with Matlab's fzero (we will develop our own ...
   algorithm below)
x0 = [1 ; 2];
xi = fsolve(f, x0);

% Jacobian of g(x) = x - lambda*f(x) at xi, as a function of ...
   lambda
Jgxi = @(lambda) eye(2) - lambda * Jf(xi);

% Plot norm(Jgxi) for a range of values of lambda, and for ...
   different norms.
figure;
hold all;
for p = [1, 2, inf]
    handle = ezplot(@(lambda) norm(Jgxi(lambda), p), [-.1, .8]);
    set(handle, 'LineWidth', 1.5);
end
ylim([.9, 1.3]);
legend('1-norm', '2-norm', '\infty-norm');
title('$\|J_g(\xi)\|_p$ for $g(x) = x - \lambda f(x)$', ...
   'Interpreter', 'Latex');
```

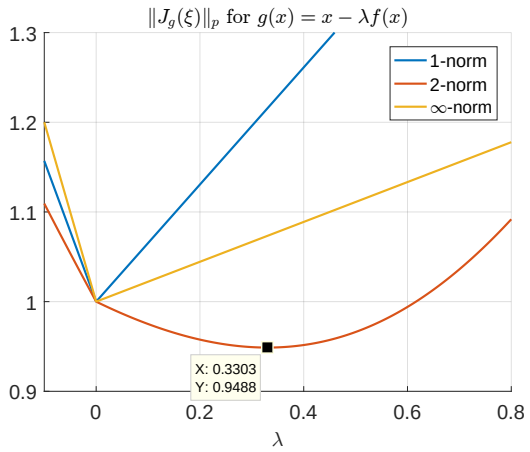


Figure 4.3: The root of f in the positive orthant is indeed a stable fixed point for g with λ between 0 and about 0.6, as indicated by the 2-norm of the Jacobian. After the fact, we find that $\lambda = 0.33$ would have been a better choice than $\lambda = 0.05$, but this is hard to know ahead of time.

A word about different norms. Notice that the contraction mapping theorem asserts convergence to a unique fixed point provided g is a contraction *in some norm*. It is important to remark that it is sufficient to have the contraction property in *one* norm to ascertain convergence. Indeed, as the example below indicates, it may be that g is not a contraction in some norm, but is a contraction in another norm. The other way around: observing $\|J_g(\xi)\| > 1$ in any number of norms is not enough to rule out convergence (contrary to the one-dimensional case where we only had to check $|g'(\xi)|$.)

```
f  = @(x) [ (x(1)^2 + x(2)^2 - 9)/9 ; -x(1) + x(2) - 1 ];
Jf = @(x) [ (2*x(1))/9, (2*x(2))/9 ; -1, 1 ];

lambda = 0.05;
g  = @(x) x - lambda*f(x);
Jg = @(x) eye(2) - lambda*Jf(x);

% Get a root with Matlab's fzero (we will develop our own ...
   algorithm below)
x0 = [1 ; 2];
xi = fsolve(f, x0);

% Display three subordinate matrix norms of the Jacobian at xi.
for p = [1, 2, inf]
    fprintf('%g-norm of Jg(xi) = %.4g\n', p, norm(Jg(xi), p));
```

```
end
```

```
  1-norm of Jg(xi) = 1.0330
  2-norm of Jg(xi) = 0.9867
Inf-norm of Jg(xi) = 1.0110
```

Thus, g is locally a contraction around the positive-orthant fixed point in the 2-norm, but not in the 1-norm or ∞-norm. That is fine: the fact it is a contraction in the 2-norm is sufficient to guarantee convergence. The large value 0.987 also supports the observation that convergence is slow in the 2-norm.

Running the same code with `xi = fsolve(f, [-1, -2])` to investigate the other fixed point yields:

```
  1-norm of Jg(xi) = 1.078
  2-norm of Jg(xi) = 1.041
Inf-norm of Jg(xi) = 1.046
```

We have $\|J_g(\xi)\| > 1$ in all three norms. We cannot conclude from this observation, but it does decrease our hope to see simple iteration with g converge to that root.

4.4 Newton's method

We now vastly broaden the class of relaxation we consider. Instead of considering relaxations of the form $g(x) = x - \lambda f(x)$ for $\lambda \neq 0$ in $\mathbb{R}$, why not allow ourselves to consider

$$g(x) = x - Mf(x),$$

where $M \in \mathbb{R}^{n\times n}$ is a nonsingular matrix? It is still true that $g(x) = x \iff f(x) = 0$ (verify it).

This generalization turns out to be beneficial even for simple examples. Indeed, consider computing roots of f below with the former scheme:

$$f(x) = \begin{bmatrix} \frac{1}{2}x_1 \\ -\frac{1}{2}x_2 \end{bmatrix}, \qquad g(x) = x - \lambda f(x),$$
$$J_f(x) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} \end{bmatrix}, \qquad J_g(x) = I_2 - \lambda J_f(x) = \begin{bmatrix} 1-\frac{\lambda}{2} & 0 \\ 0 & 1+\frac{\lambda}{2} \end{bmatrix}.$$

At any point x, for any choice of λ, the norm of the Jacobian is at least 1 in the 2-norm (and also in the 1-norm and in the ∞-norm). Yet, if we allow ourselves to use a matrix M, then setting $M = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ yields:

$$g(x) = x - Mf(x), \qquad J_g(x) = I_2 - MJ_f(x) = \frac{1}{2} I_2.$$

The 2-norm (and 1-norm, and ∞-norm) of this matrix is $\frac{1}{2} < 1$, hence, local convergence is guaranteed.

Based on our understanding of the critical role of a small Jacobian at ξ, ideally, we would pick M such that

$$0 = J_g(\xi) = I_n - MJ_f(\xi), \text{ hence,} \qquad M = (J_f(\xi))^{-1},$$

assuming the inverse exists. Of course, we do not know ξ, let alone $(J_f(\xi))^{-1}$. Reasoning that at iteration k our best approximation for ξ is $x^{(k)}$ (as far as we know at that point), we are lead to Newton's method:

Definition 4.6 (Newton's method in $\mathbb{R}^n$)**.** *For a differentiable function f, given $x^{(0)} \in \mathbb{R}^n$, Newton's method generates the sequence*

$$x^{(k+1)} = x^{(k)} - (J_f(x^{(k)}))^{-1} f(x^{(k)}), \qquad k = 0, 1, 2, \ldots$$

This is equivalent to simultaneous iteration with

$$g(x) = x - (J_f(x))^{-1} f(x).$$

It is implicitly assumed all Jacobians encountered are nonsingular.

One important remark is that one should *not* compute the inverse of $J_f(x^{(k)})$. Instead, solve the linear system implicitly defined by the iteration (using LU with pivoting for example, possibly via Matlab's backslash):

$$\underbrace{J_f(x^{(k)})}_{A} \underbrace{(x^{(k+1)} - x^{(k)})}_{x} = \underbrace{-f(x^{(k)})}_{b}.$$

Add the solution to $x^{(k)}$ to obtain $x^{(k+1)}$. This is faster and numerically more accurate than computing the inverse of the matrix.

Importantly, with Newton's method, we can get any root of f where the Jacobian is nonsingular. On our original example (intersection of circle and line), initializing Newton's method at various points can yield convergence to either root.

```
f = @(x) [ (x(1)^2 + x(2)^2 - 9)/9 ; -x(1) + x(2) - 1 ];
Jf = @(x) [ 2*x(1)/9, 2*x(2)/9 ; -1, 1 ];

x = [1 ; 2];  % initialize

for k = 1 : 8

    fx  = f(x);
    Jfx = Jf(x);

    x = x - (Jfx\fx); % Newton's step: solve a linear system

    fprintf('x = [%+.2e, %+.2e],  f(x) = [%+.2e, %+.2e];\n', ...
              x(1), x(2), fx(1), fx(2));

end
```

```
x = [+1.67e+00, +2.67e+00],  f(x) = [-4.44e-01, +0.00e+00];
x = [+1.56e+00, +2.56e+00],  f(x) = [+9.88e-02, +0.00e+00];
x = [+1.56e+00, +2.56e+00],  f(x) = [+2.34e-03, +2.22e-16];
x = [+1.56e+00, +2.56e+00],  f(x) = [+1.44e-06, -2.22e-16];
x = [+1.56e+00, +2.56e+00],  f(x) = [+5.51e-13, -2.22e-16];
x = [+1.56e+00, +2.56e+00],  f(x) = [+0.00e+00, +0.00e+00];
x = [+1.56e+00, +2.56e+00],  f(x) = [+0.00e+00, +0.00e+00];
x = [+1.56e+00, +2.56e+00],  f(x) = [+0.00e+00, +0.00e+00];
```

Here is the output if we initialize at $(-1,-2)^T$:

```
x = [-3.00e+00, -2.00e+00],  f(x) = [-4.44e-01, -2.00e+00];
x = [-2.60e+00, -1.60e+00],  f(x) = [+4.44e-01, +0.00e+00];
x = [-2.56e+00, -1.56e+00],  f(x) = [+3.56e-02, +0.00e+00];
x = [-2.56e+00, -1.56e+00],  f(x) = [+3.22e-04, +2.22e-16];
x = [-2.56e+00, -1.56e+00],  f(x) = [+2.75e-08, -2.22e-16];
x = [-2.56e+00, -1.56e+00],  f(x) = [+1.97e-16, -2.22e-16];
x = [-2.56e+00, -1.56e+00],  f(x) = [+0.00e+00, +2.22e-16];
x = [-2.56e+00, -1.56e+00],  f(x) = [+0.00e+00, +0.00e+00];
```

Convergence is fast! We will prove *quadratic* convergence momentarily. Here is what that means in $\mathbb{R}^n$ [SM03, Def. 4.6]

Definition 4.7. *Suppose $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ converges to $\xi \in \mathbb{R}^n$. We say the sequence converges* with at least order $q > 1$ *if the errors $\|x^{(k)} - \xi\|$ converge*

to 0 with at least order q, that is, if there exists a sequence $\varepsilon_0, \varepsilon_1, \ldots > 0$ converging to zero and $\mu \geq 0$ such that

$$\forall k,\ \|x^{(k)} - \xi\| \leq \varepsilon_k, \qquad \text{and} \qquad \lim_{k\to\infty} \frac{\varepsilon_{k+1}}{\varepsilon_k^q} = \mu,$$

for some arbitrary norm in $\mathbb{R}^n$. We similarly define convergence with order q, at least quadratic convergence, and quadratic convergence by applying those definitions to the sequence of error norms $\|x^{(k)} - \xi\|$.

The main theorem about Newton's method follows [SM03, Thm. 4.4].

Theorem 4.8. *Let $f\colon \mathbb{R}^n \to \mathbb{R}^n$ be three times continuously differentiable, and let ξ be a root of f. Assume $J_f(\xi)$ is nonsingular. Then, provided $x^{(0)}$ is sufficiently close to ξ, Newton's method converges to ξ at least quadratically.*

Proof. The proof is in two parts. First, we establish convergence. Then, we establish the speed of convergence.

Convergence. At a high level, we really only need to investigate the norm of $J_g(\xi)$. Since $g(x) = x - (J_f(x))^{-1} f(x)$, the chain rule yields:

$$J_g(x) = I_n - \left[\ \text{the derivative of } (J_f(x))^{-1}\ \right] f(x) - (J_f(x))^{-1} J_f(x),$$

where we are not too precise about the term in the bracket for now. From this and the fact that $f(\xi) = 0$, we deduce that $J_g(\xi) = 0$. We now give further details about this computation. Define $A(x) = (J_f(x))^{-1}$. This is a differentiable function of x at ξ.[2] Since $g(x) = x - A(x)f(x)$, we have:

This is similar to the observation that $x \mapsto \frac{1}{a(x)}$ is differentiable at x provided $a(x) \neq 0$ and a is differentiable.

$$\begin{aligned} g_i(x) &= x_i - \sum_{k=1}^{n} a_{ik}(x) f_k(x), \\ \frac{\partial g_i}{\partial x_j}(x) &= \delta_{ij} - \sum_{k=1}^{n} \left[\frac{\partial a_{ik}}{\partial x_j}(x) f_k(x) + a_{ik}(x) \frac{\partial f_k}{\partial x_j}(x) \right], \end{aligned}$$

where we used the *Kronecker delta* notation:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

[2] To see this, notice first that $J_f(x)$ is a differentiable function of x by assumption. By Cramer's rule, the inverse of $J_f(x)$ is a matrix whose entries are polynomials (determinants) of the entries of $J_f(x)$, divided by the determinant of $J_f(x)$. Provided $J_f(x)$ is invertible, this is indeed differentiable in x.

The term $\sum_{k=1}^n \frac{\partial a_{ik}}{\partial x_j}(x) f_k(x)$ (thankfully) vanishes at $x = \xi$ since $f(\xi) = 0$. Focus on the other term, which is nothing but the inner product between the ith row of $A(x)$ and the j column of $J_f(x)$:

$$\begin{aligned}\sum_{k=1}^n a_{ik}(x)\frac{\partial f_k}{\partial x_j}(x) &= \sum_{k=1}^n [A(x)]_{ik}\,[J_f(x)]_{kj} \\ &= [A(x)J_f(x)]_{ij} \\ &= \left[(J_f(x))^{-1}J_f(x)\right]_{ij} \\ &= [I_n]_{ij} \\ &= \delta_{ij}.\end{aligned}$$

Thus, $J_g(\xi) = 0$, as expected. Regardless of choice of norm, $\|J_g(\xi)\| = 0 < 1$, which by our earlier considerations implies there exists a neighborhood of ξ such that if $x^{(0)}$ is in that neighborhood, then Newton's method converges to ξ.

In [SM03, Thm. 4.4], the proof is organized differently, so that it is f and not g that one expands in a Taylor series. This has the advantage that they only need f to be twice continuously differentiable. Here, since we need g to be twice continuously differentiable, we need f to be three times continuously differentiable.

More explicitly, consider (4.1) again: there exist constants $c, \delta > 0$ such that, provided $\|h\| \le \delta$,

$$\begin{aligned}g(\xi + h) &= g(\xi) + J_g(\xi)h + E(\xi, h) \\ &= g(\xi) + E(\xi, h),\end{aligned}$$

where $\|E(\xi, h)\| \le c\|h\|^2$. With $h = x^{(k)} - \xi$, if $\|x^{(k)} - \xi\| \le \delta$,

$$\begin{aligned}\|x^{(k+1)} - \xi\| &= \|g(x^{(k)}) - g(\xi)\| \\ &= \|g(\xi + h) - g(\xi)\| \\ &= \|E(\xi, h)\| \\ &\le c\|h\|^2 \\ &= c\|x^{(k)} - \xi\|^2. \end{aligned} \tag{4.2}$$

In particular, if $c\|x^{(k)} - \xi\| \le \frac{1}{2}$ (we could also have chosen another constant strictly less than 1), it follows that $\|x^{(k+1)} - \xi\| \le \frac{1}{2}\|x^{(k)} - \xi\|$. Thus, if $\|x^{(0)} - \xi\| \le \min\left(\delta, \frac{1}{2c}\right)$, then the same is true of all $x^{(k)}$ (why?) and Newton's method converges to ξ (at least linearly).[3]

[3] We omitted to verify that Newton's equation is well defined, that is, that $J_f(x^{(k)})$ is invertible for all k under some conditions on $x^{(0)}$. The argument goes as follows: $J_f(x)$ is assumed to be a continuous function of x, hence its determinant is a continuous function of x. Since the determinant is nonzero at ξ by assumption, it must be nonzero in a neighborhood of ξ by continuity, hence: Newton's method is well defined in that neighborhood. Details are in [SM03, Thm. 4.4]: we omit them for simplicity.

At least quadratic convergence. Consider (4.2) again. Under the condition above on $x^{(0)}$, it holds for all k that

$$\frac{\|x^{(k+1)} - \xi\|}{\|x^{(k)} - \xi\|^2} \leq c.$$

Thus, taking the limit,

$$\lim_{k\to\infty} \frac{\|x^{(k+1)} - \xi\|}{\|x^{(k)} - \xi\|^2} \leq c.$$

This establishes at least quadratic convergence. (Note: this convergence is quadratic if the limit *equals* a positive number; if it equals 0 (which is not excluded here), then the convergence is faster than quadratic.) □

A word to conclude. It is a common misconception that the above theorem allows this conclusion: "Newton's method converges to the root which is closest to $x^{(0)}$." This is completely false, in so many ways:

1. What does "closest" mean? In what norm?
2. The above theorem does *not* guarantee convergence *in general*.
3. Rather, it says: around each nondegenerate[4] root ξ, there is a (possibly very small) neighborhood of ξ such that initializing in that neighborhood guarantees (fast) convergence to that root.
4. Initializing outside that neighborhood, anything can happen: convergence to ξ anyway (possibly slow at first), convergence to another root (possibly far away), or even divergence.

Consider Figure 4.4, known as a Newton fractal, to anchor this remark. Despite this warning, Newton's is one of the most useful algorithms in numerical analysis, as it allows to refine crude approximations to high accuracy.

[4]That is, such that the Jacobian of f at that root is nonsingular.

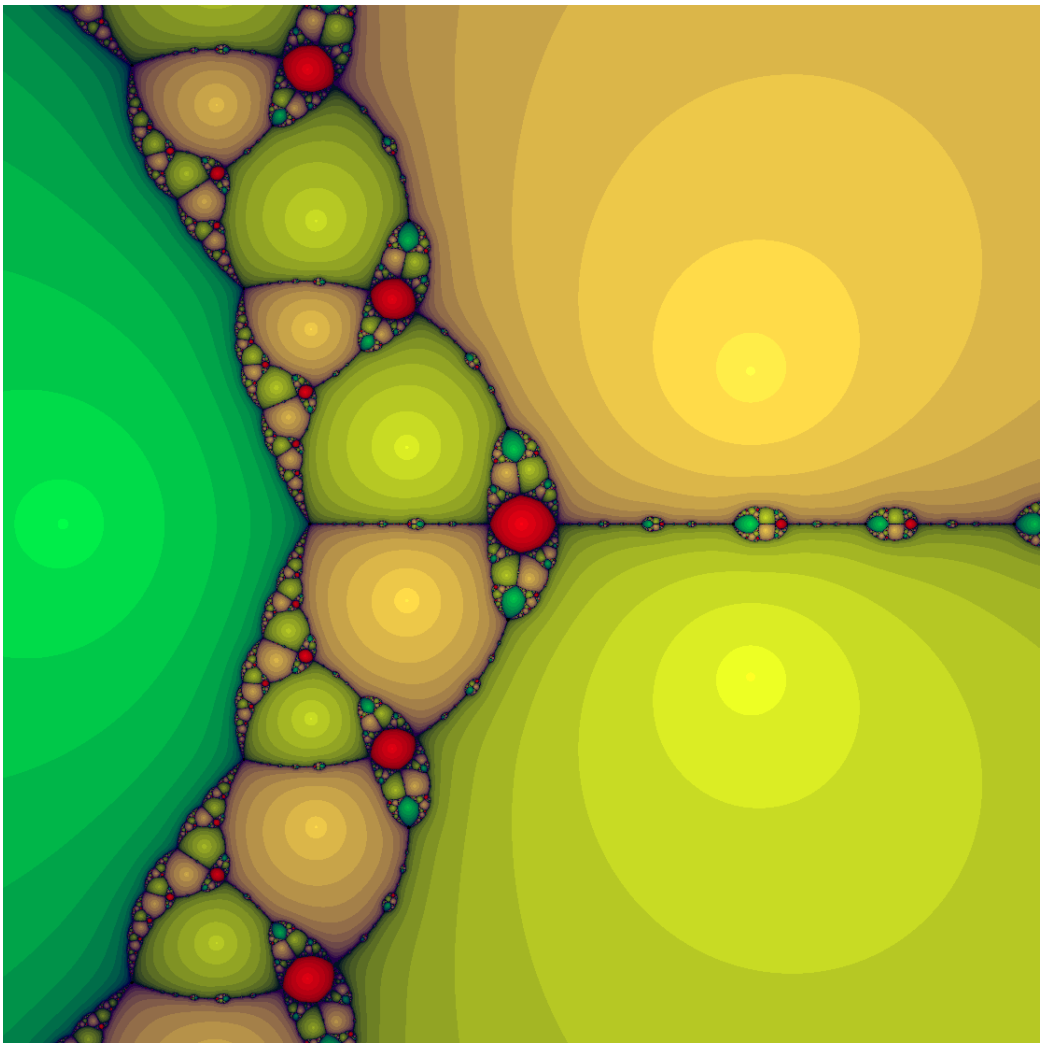


Figure 4.4: Figure by Henning Makholm, available at https://en.wikipedia.org/wiki/Newton_fractal. Newton fractal generated by the polynomial $f(z) = z^3 - 2z + 2$, where $f\colon \mathbb{C} \to \mathbb{C}$ can also be thought of as a function $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ by separating real and imaginary part. The image area covers a square with side length 4 centered on 0. Corner pixels represent $\pm 2 \pm 2i$ exactly. The three roots are given by `roots([1 0 -2 2])` as `-1.7693 + 0.0000i, 0.8846 + 0.5897i, 0.8846 - 0.5897i`. The three shades of yellow/green pixel colors indicates which root Newton converges to if initialized there. Pixel intensity indicates the number of iterations it takes to get closer than 10^{-3} (in complex absolute value) to a root. Red color suggests no convergence (divergence or cycling). Notice how sensitive the limit point can be to small changes in initialization.

Chapter 5

Computing eigenvectors and eigenvalues

In this chapter, we study algorithms to compute eigenvectors and eigenvalues of matrices. We follow a mixture of Chapter 5 in [SM03], Lectures 24–27 and 30–31 in [TBI97], and some extra material.

Problem 5.1. *Given a matrix $A \in \mathbb{R}^{n\times n}$ (or in $\mathbb{C}^{n\times n}$), an eigenpair of A is a pair (λ, x) such that $x \in \mathbb{C}^n$ is nonzero, $\lambda \in \mathbb{C}$, and*

$$Ax = \lambda x.$$

The eigenproblem is that of computing one or all eigenpairs of A.

Notice that

$$\begin{aligned}
&\lambda \text{ is an eigenvalue of } A \\
\iff &Ax = \lambda x \text{ has a solution } x \neq 0 \\
\iff &(A - \lambda I_n)x = 0 \text{ has a solution } x \neq 0 \\
\iff &A - \lambda I_n \text{ is not invertible} \\
\iff &\det(A - \lambda I_n) = 0.
\end{aligned}$$

This leads to the notion of characteristic polynomial.

Definition 5.2. *Given a square matrix A of size n, the characteristic polynomial of A, defined by*

$$p_A(\lambda) = \det(A - \lambda I_n)$$

is a polynomial of degree n whose roots are the eigenvalues of A. Thus, by the fundamental theorem of algebra, A always has exactly n eigenvalues in the complex plane (counting multiplicities).

Example 5.3. *For example consider* $A = \begin{bmatrix} 2 & 1 \\ -3 & 3 \end{bmatrix}$. *Its characteristic polynomial is*

$$p_A(\lambda) = \det(A - \lambda I_2) = \det\left(\begin{bmatrix} 2-\lambda & 1 \\ -3 & 3-\lambda \end{bmatrix}\right) = \lambda^2 - 5\lambda + 9.$$

Its two complex roots are $2.5 \pm i1.658\ldots$*: these are the eigenvalues of* A*.*

Remark 5.4. *Matlab provides the function* `poly` *which returns the coefficients of* p_A *in the monomial basis. Notice that Matlab defines the characteristic polynomial as* $\det(\lambda I_n - A)$*, which is equivalent to* $(-1)^n p_A(\lambda)$*. Of course, this does not change the roots.*

Remark 5.5. *Owing to work by Abel (later complemented by Galois), it is known since the 19th century that there does not exist any closed-form formula for the roots of a polynomial of degree five or more using only basic arithmetic operations* $(+, -, \times, /, \sqrt{\cdot})$*. Furthermore, for any polynomial* p *of degree* n*, one can easily construct a matrix* A *of size* n *such that* $p_A = p$ *(a so-called* companion matrix*). Consequently, no finite-time algorithm can compute the eigenvalues of a general matrix of size* $n \geq 5$*: in this chapter, we* must *use iterative algorithms. This is in stark contrast with other linear algebra problems we covered such as solving* $Ax = b$ *and factoring* $A = QR$*.*

One obvious idea to compute the eigenvalues of A is to obtain the characteristic polynomial, that is, compute $a_0, \ldots, a_n$ such that

$$p_A(\lambda) = a_n\lambda^n + \cdots + a_1\lambda + a_0,$$

then to compute the roots of p_A (for example, using the iterative algorithms we covered to solve nonlinear equations.) This turns out to be a terrible idea.

```
n = 50; % try 50, 200, 400
A = randn(n)/sqrt(n); % scaling to (mostly) keep eigs in ...
   disk of radius 1.5

p = poly(A);          % The problem is already here,
d1 = roots(p);        % not so much here.

d2 = eig(A);          % We can trust this one.

subplot(1,2,1);
plot(real(d1), imag(d1), 'o', 'MarkerSize', 8); hold all;
plot(real(d2), imag(d2), 'x', 'MarkerSize', 8); hold off;
xlim([-1.3, 1.3]);
```

```
ylim([-1.3, 1.3]);
pbaspect([1,1,1]);
title('Eigenvalues of A and computed roots of p_A');
xlabel('Real part');
ylabel('Imaginary part');
legend('Computed roots', '"True" eigenvalues');

subplot(1,2,2);
stem(n:-1:0, abs(p));
pbaspect([1.6,1,1]);
xlim([0, n]);
set(gca, 'YScale', 'log');
title('Coefficients of p_A(\lambda)');
xlabel('Absolute coefficient of \lambda^k for k = 0...n');
ylabel('Absolute value of coefficient');
```

Running the code with $n = 50$ and 400 produces Figures 5.1 and 5.2. As can be seen from the figures, the coefficients of p_A in the monomial basis $(1, \lambda, \lambda^2, \ldots)$ grow out of proportions already for small values of n: there are many orders of magnitude between them, far more than 16. Even if we find an excellent algorithm to compute the coefficients in IEEE arithmetic and to compute the roots of the polynomial from there, the round-off error on the coefficients alone is already too much to preserve a semblance of accuracy. This is because the conditioning of the problem "given the coefficients of a polynomial in the monomial basis, find the roots of that polynomial" is very bad. In other words: small perturbations of the coefficients may lead to large perturbations of the roots. We claim this here without proof. We will develop other ways.

Remark 5.6. *Computing the eigenvalues of A by first computing its characteristic polynomial p_A is a terrible idea. Yet, the other way around is a good idea: given a polynomial p, building a so-called* companion *matrix A such that p_A and p have the same roots, then using an eigenproblem algorithm to compute the eigenvalues of A is a good strategy to compute the roots of p. Type* `edit roots` *in Matlab to see that this is the strategy used by Matlab.*

5.1 The power method

In this section, we discuss one of the simplest algorithms available to compute the largest eigenvalue (in magnitude) of a matrix $A \in \mathbb{C}^{n\times n}$, and an associated eigenvector. For convenience, we assume A is diagonalizable, which is the case for almost all matrices in $\mathbb{C}^{n\times n}$. That is, we assume there exists

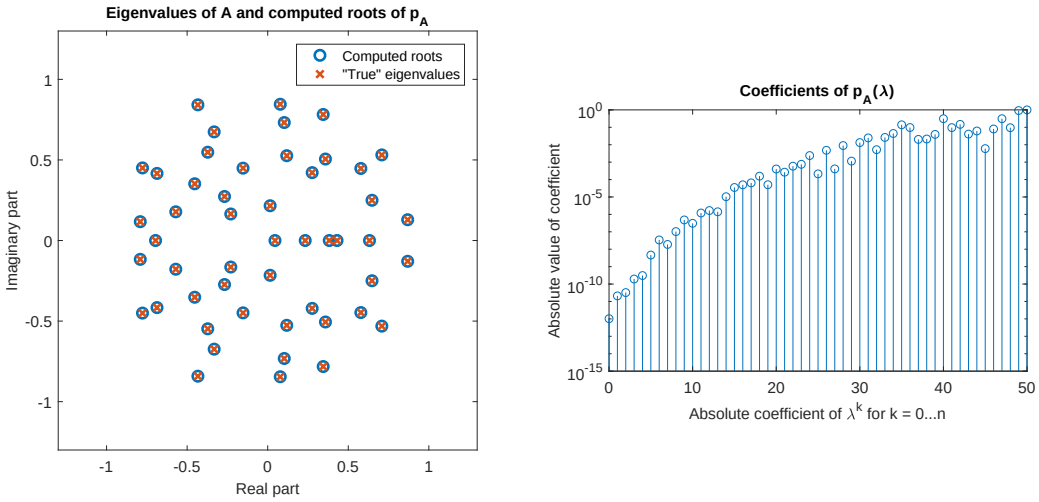


Figure 5.1: Computing the eigenvalues of A as the roots of its characteristic polynomial p_A with $n = 50$. Left: actual roots and computed roots in complex plane; Right: coefficients of p_A in absolute value (logarithmic scale). The "true" eigenvalues are those computed by Matlab's `eig`: they are trustworthy.

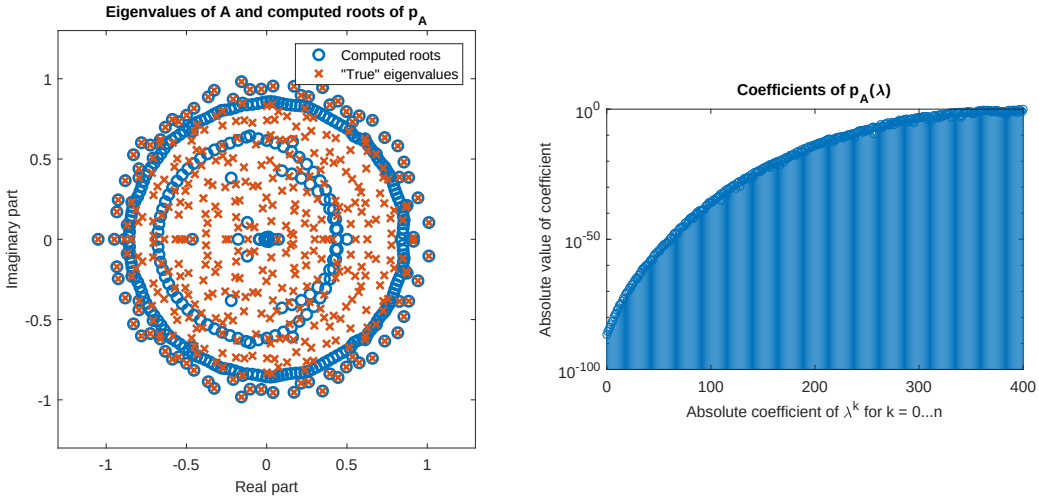


Figure 5.2: Same as Figure 5.1 with $n = 400$: the computed roots are completely off. The strange pattern of the roots of the computed characteristic polynomial is not too relevant for us. As a side note on that topic, notice that the roots of a polynomial with random Gaussian coefficients follow the same kind of pattern: `x = roots(randn(401, 1)); plot(real(x), imag(x), '.'); axis equal;`.

$V \in \mathbb{C}^{n\times n}$ invertible and $D \in \mathbb{C}^{n\times n}$ diagonal such that

$$A = VDV^{-1}.$$

In other words,

$$AV = VD,$$

which implies $D = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ contains the eigenvalues of A and v_k—the kth column of V—is an eigenvector associated to λ_k. Without loss of generality, we can assume $\|v_k\|_2 = 1$ for all k.

Assumption 0 *A is diagonalizable.*

Let the eigenvalues be ordered in such a way that

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|. \tag{5.1}$$

Thus, λ_1 is a largest eigenvalue in magnitude, with eigenvector v_1: these are our targets.

The power iteration is a type of simultaneous iteration. For a given nonzero $x^{(0)} \in \mathbb{C}^n$, it generates a sequence of iterates as follows:

$$x^{(k+1)} = g(x^{(k)}), \text{ for } k = 0, 1, 2, \ldots, \text{ where} \tag{5.2}$$

$$g(x) = \frac{Ax}{\|Ax\|_2}. \tag{5.3}$$

The 2-norm for complex vectors is defined by $\|u\|_2 = \sqrt{u^*u} = \sqrt{\sum_{i=1}^n |u_i|^2}$, where u^* denotes the complex Hermitian conjugate-transpose of a vector (or matrix).

Question 5.7. *Show that the fixed points of g are eigenvectors of A.*

Question 5.8. *Show by induction that $x^{(k)} = \frac{A^k x^{(0)}}{\|A^k x^{(0)}\|_2}$ for $k = 1, 2, \ldots$*

Thus, to understand the behavior of the power iteration, we must understand the behavior of $A^k x^{(0)}$. Using the relation $A = VDV^{-1}$, we can establish the following:

$$\begin{aligned} A^k &= (VDV^{-1})(VDV^{-1})\cdots(VDV^{-1}) \\ &= VD(V^{-1}V)D(V^{-1}V)D\cdots(V^{-1}V)DV^{-1} \\ &= VD^kV^{-1}. \end{aligned} \tag{5.4}$$

Equivalently:

$$A^k V = V D^k, \tag{5.5}$$

where the matrix D^k is diagonal, with entries $\lambda_1^k, \ldots, \lambda_n^k$.

The initial iterate $x^{(0)}$ can always be expanded in the basis of eigenvectors $v_1, \ldots, v_n$ with some coefficients $c_1, \ldots, c_n \in \mathbb{C}$, so that

$$x^{(0)} = c_1 v_1 + \cdots + c_n v_n = Vc. \tag{5.6}$$

Thus,

$$A^k x^{(0)} = A^k V c = V D^k c = (c_1 \lambda_1^k) v_1 + \cdots + (c_n \lambda_n^k) v_n. \tag{5.7}$$

Let us factor out the (complex) number λ_1^k from this expression:

$$A^k x^{(0)} = \lambda_1^k \left(c_1 v_1 + \sum_{j=2}^{n} c_j \left(\frac{\lambda_j}{\lambda_1} \right)^k v_j \right). \tag{5.8}$$

Now come the crucial assumptions for the power method:

Assumption 1 The largest eigenvalue is strictly larger than all others: $|\lambda_1| > |\lambda_2|$.

Assumption 2 The initial iterate $x^{(0)}$ aligns at least somewhat with v_1, that is: $c_1 \neq 0$.[1]

Under these conditions, it is clear that $(\lambda_j/\lambda_1)^k \to 0$ as $k \to \infty$ for $j = 2, \ldots, n$, so that, asymptotically, $A^k x^{(0)}$ is aligned with v_1: the dominant eigenvector. More precisely,

$$x^{(k)} = \frac{A^k x^{(0)}}{\|A^k x^{(0)}\|_2} = \frac{\lambda_1^k}{|\lambda_1^k|} \frac{c_1 v_1 + \sum_{j=2}^{n} c_j \left(\frac{\lambda_j}{\lambda_1} \right)^k v_j}{\left\| c_1 v_1 + \sum_{j=2}^{n} c_j \left(\frac{\lambda_j}{\lambda_1} \right)^k v_j \right\|_2}.$$

If λ_1 is a positive real number, then $\frac{\lambda_1^k}{|\lambda_1^k|} = 1$ for all k so that the limit exists:

Here, we use $\|v_1\|_2 = 1$.

$$\lim_{k \to \infty} x^{(k)} = \lim_{k \to \infty} \frac{c_1 v_1}{\|c_1 v_1\|_2} = \frac{c_1}{|c_1|} v_1. \tag{5.9}$$

[1] If A is Hermitian ($A = A^*$), this is equivalent to saying: $x^{(0)}$ is not orthogonal to v_1. For a general matrix, it is equivalent to saying: $x^{(0)}$ does not lie in the subspace spanned by $v_2, \ldots, v_n$. If $x^{(0)}$ is taken as a (complex) Gaussian random vector, this is satisfied with probability 1.

The complex number $\frac{c_1}{|c_1|}$ has modulus one: its presence is indicative of the fact that unit-norm eigenvectors are defined only *up to phase*: if v_1 is an eigenvector, then so are $-v_1$, iv_1, $-iv_1$ and all other vectors of the form $e^{i\theta}v_1$ for any $\theta \in \mathbb{R}$. The important point is that $x^{(k)}$ asymptotically aligns with v_1, up to an unimportant complex phase. More generally, if λ_1 is not a positive real number, then the limit of $x^{(k)}$ does not exist because the phase may keep changing due to the term $\frac{\lambda_1^k}{|\lambda_1^k|}$; this is inconsequential as it does not affect the direction spanned by $x^{(k)}$.[2]

The convergence rate is dictated by the ratio $\frac{|\lambda_2|}{|\lambda_1|}$. If this is close to 1, convergence is slow; if this is close to 0, convergence is fast. In all cases, unless this ratio is zero, the convergence is linear since the error decays as $\left(\frac{|\lambda_2|}{|\lambda_1|}\right)^k$ times a constant.

If after k iterations $v = x^{(k)}$ is deemed a reasonable approximate dominant eigenvector of A, we can extract an approximate corresponding eigenvalue using the *Rayleigh quotient*:

$$\begin{aligned} Av &\approx \lambda_1 v, \text{ thus} \\ v^*Av &\approx \lambda_1 v^*v, \text{ so} \\ \lambda_1 &\approx \frac{v^*Av}{\|v\|_2^2}. \end{aligned} \tag{5.10}$$

For A real and symmetric, see Theorem 5.13 in [SM03] or Theorem 27.1 in [TBI97] for guarantees on the approximation quality.

5.2 Inverse iteration

The power method is nice, but it has two main shortcomings:

1. It can only converge to a dominant eigenpair; and
2. Convergence speed is dictated by $\left|\frac{\lambda_1}{\lambda_2}\right|$: it could be slow.

The central idea behind *inverse iteration* is to use the power method on a different matrix to address both these issues.

For a diagonalizable matrix $A = VDV^{-1}$ and a given $\mu \in \mathbb{C}$, observe the following:

$$A - \mu I_n = VDV^{-1} - \mu VV^{-1} = V(D - \mu I_n)V^{-1}.$$

[2] In comparison, observe that $\lim_{k\to\infty} x^{(k)}(x^{(k)})^* = v_1 v_1^*$: this limit exists even if λ_1 is not real positive.

The right-hand side is a diagonalization of the left-hand side; thus: the eigenvalues of $A - \mu I_n$ are $\lambda_1 - \mu, \ldots, \lambda_n - \mu$, and both A and $A - \mu I_n$ have the same eigenvectors. Now, take the inverse:

$$(A - \mu I_n)^{-1} = V(D - \mu I_n)^{-1} V^{-1} = V \begin{pmatrix} \frac{1}{\lambda_1 - \mu} & & \\ & \ddots & \\ & & \frac{1}{\lambda_n - \mu} \end{pmatrix} V^{-1}. \quad (5.11)$$

Thus, $(A - \mu I_n)^{-1}$ shares eigenvectors with A, and the eigenvalues of $(A - \mu I_n)^{-1}$ are $\frac{1}{\lambda_1 - \mu}, \cdots, \frac{1}{\lambda_n - \mu}$. Hence, if we know how to pick μ closer to a desired *simple* eigenvalue λ_ℓ than to any other, then running the power method on $(A - \mu I_n)^{-1}$ will produce convergence to an eigenvector associated to λ_ℓ. Indeed: $\frac{1}{\lambda_\ell - \mu}$ is then the largest eigenvalue in absolute value. Furthermore, if we are really good at picking μ close to λ_ℓ, we can get an excellent linear convergence rate.

For a given $A \in \mathbb{C}^{n \times n}$, $\mu \in \mathbb{C}$ and $x^{(0)} \in \mathbb{C}^n$ (nonzero, often taken at random), the inverse power iteration executes the following:

$$y^{(k+1)} = (A - \mu I_n)^{-1} x^{(k)}, \quad (5.12)$$

$$x^{(k+1)} = \frac{y^{(k+1)}}{\|y^{(k+1)}\|_2}. \quad (5.13)$$

Each iteration involves solving a system of linear equations in $M = A - \mu I_n$. Since this is the same matrix over and over again, it pays to do the following:

1. Compute the LU factorization of M, that is $PM = LU$ *once* for $O(n^3)$ flops, then

2. Once per iteration, solve the system $My^{(k+1)} = x^{(k)}$ using the LU factorization for only $O(n^2)$ flops each time.

Question 5.9. *Compare the costs of inverse iteration and the power method.*

Numerically, one aspect of inverse iteration screams for attention. Indeed, it appears that we have an incentive to pick μ as close as possible to λ_ℓ; yet, this will undoubtedly deteriorate the condition number of $A - \mu I_n$. In fact, if we manage to set $\mu = \lambda_\ell$—which seems ideal—then the matrix is not even invertible. How could one hope to solve the linear system (5.12) with good accuracy? It turns out that this is not a problem.

Reconsider the linear system to be solved in inverse iteration:

$$(A - \mu I_n)y = b.$$

If b is perturbed to become $b + \delta b$, then y is perturbed as well to $y + \delta y$ and these perturbations are related by:

$$(A - \mu I_n)\delta y = \delta b.$$

Using $A = VDV^{-1}$, remember we have

$$(A - \mu I_n)^{-1} = V(D - \mu I_n)^{-1}V^{-1}.$$

Define the vector $u = V^{-1}\delta b$ with entries $u_1, \ldots, u_n$. Then,

$$\begin{aligned}\delta y &= (A - \mu I_n)^{-1}\delta b \\ &= V(D - \mu I_n)^{-1}V^{-1}\delta b \\ &= V(D - \mu I_n)^{-1}u \\ &= \sum_{i=1}^{n} \frac{u_i}{\lambda_i - \mu} v_i.\end{aligned}$$

If $\mu \approx \lambda_\ell$ for some particular ℓ (and μ is significantly different from all other eigenvalues), the corresponding term in this sum dominates and we get

$$\delta y \approx \frac{u_\ell}{\lambda_\ell - \mu} v_\ell.$$

Crucially, this perturbation δy in the solution y is primarily aligned with v_ℓ; but v_ℓ is exactly what we intend to compute! As inverse iteration converges, y aligns with v_ℓ, and so does most of the perturbation δy. Upon normalizing y to get the next iterate, the effects of ill-conditioning are essentially forgotten.[3]

5.3 Rayleigh quotient iteration

One point in particular can be much improved regarding inverse iteration: the choice of μ, which is supposed to approximate λ_ℓ. First of all, it is not clear that one can always know a good value for μ ahead of time. Second, it makes sense that, as we iterate and $x^{(k)}$ converges, we should be able to exploit that to improve μ. The Rayleigh quotient, already defined in (5.10), helps do exactly that.

[3] IEEE arithmetic can still run into trouble here, specifically due to under/overflow. If this happens, `Inf`'s and `NaN`'s will appear in the solution y: the reader should think carefully about what to do when that happens.

The main idea behind *Rayleigh quotient iteration* (RQI) is to use the Rayleigh quotient at each iteration to redefine μ. Given A and $x^{(0)}$ (with the latter often chosen at random), RQI iterates as follows:

$$\mu_k = \frac{(x^{(k)})^T A x^{(k)}}{(x^{(k)})^T x^{(k)}}, \tag{5.14}$$

$$y^{(k+1)} = (A - \mu_k I_n)^{-1} x^{(k)}, \tag{5.15}$$

$$x^{(k+1)} = \frac{y^{(k+1)}}{\|y^{(k+1)}\|_2}. \tag{5.16}$$

(You can also choose to initialize μ_0 independently of $x^{(0)}$ to favor convergence to an eigenpair with eigenvalue close to μ_0: think about it.)

Note that, contrary to the situation for inverse iteration, it is not clear if and where to RQI converges: we do not analyze it in this course. When RQI converges, for real symmetric matrices (or complex Hermitian), it often does so *cubically*. This is extremely fast. On the other hand, now μ_k changes at each iteration, which means we can no longer precompute an LU factorization of the linear system once and for all: it seems as if each iteration must cost $O(n^3)$ flops.

Fortunately, this computational burden can be reduced by transforming A to a so-called *Hessenberg form*.[4] In this course, we only focus on a particular case: when A is real and symmetric, transforming it to Hessenberg form means: to make it tridiagonal. We discuss algorithms for this task later in this chapter. For $O(n^3)$ flops, such algorithms produce an orthogonal matrix P and a symmetric, tridiagonal matrix H such that

$$A = PHP^T.$$

In Matlab, this is built-in as `[P, H] = hess(A)`.

To understand how the eigenvectors and eigenvalues of A relate to those of the simpler matrix H, it is useful to remember the spectral theorem: if A is real and symmetric,

1. A is diagonalizable with *real* eigenvalues and *real* eigenvectors; and
2. The eigenvectors $v_1, \ldots, v_n$ can be taken to form an *orthonormal* basis.

In other words, there exists $D = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ (real) and $V \in \mathbb{R}^{n \times n}$ *orthogonal* such that

$$A = VDV^T, \tag{5.17}$$

[4] A matrix is in Hessenberg form if $A_{ij} \neq 0 \implies j \geq i + 1$.

and $\lambda_1 \geq \cdots \geq \lambda_n$ (where the ordering makes sense because the eigenvalues are real).

Question 5.10. *Show that if (λ, x) is an eigenpair of H, then (λ, Px) is an eigenpair of A.*

It is left to you as an exercise to figure out how best to use tridiagonalization in conjunction with RQI to reduce the cost of each iteration to a mere $O(n)$ flops. Whether or not the precomputation is worth the effort may depend on the total number of RQI iterations you intend to run with A.

Remark 5.11. *In treating the power method, inverse iteration and RQI, we never acknowledged the possibility of the targeted eigenvalue having (geometric) multiplicity greater than 1. Analyzing these methods in that scenario takes more care, beyond the scope of this introductory course. You are encouraged to experiment numerically to get a sense of how these methods behave in that case.*

5.4 Eigenvalues of a symmetric, tridiagonal matrix: Sturm sequences

When we approached solving $Ax = b$, we considered the simplified problem of solving a triangular system first, then showed how one can reduce any system to a pair of triangular systems. In the same spirit, we now consider the particular problem of computing the eigenvalues of a symmetric, tridiagonal matrix, and later we will show how to reduce any symmetric matrix to a tridiagonal one without changing its eigenvalues.

Problem 5.12. *Given a symmetric, tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, compute some or all of its eigenvalues $\lambda_n \leq \cdots \leq \lambda_1$.*

Notice that we do not care about the eigenvectors for now. Using inverse iteration or RQI, you should be able to recover eigenvectors efficiently after computing eigenvalues.

Let the matrix T be defined as:

$$T = \begin{bmatrix} a_1 & b_2 & & & \\ b_2 & a_2 & b_3 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-1} & a_{n-1} & b_n \\ & & & b_n & a_n \end{bmatrix}. \tag{5.18}$$

Without loss of generality, we can make the following assumption:

Assumption 5.13. *For all $i = 2, \ldots, n$, assume $b_i \neq 0$.*

Indeed, if any $b_k = 0$, the eigenvalue problem separates into smaller ones with the same structure.

Question 5.14. *Let $M = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}$ be a matrix with blocks A, B (square). Show that the eigenvalues of A are eigenvalues of M. Similarly, show that the eigenvalues of B are eigenvalues of M. Deduce that the set of eigenvalues of M is exactly the union of the sets of eigenvalues of A and B.*

■

Question 5.15. *What can you tell about the eigenvalues of T if $b_2 = 0$? If $b_3 = 0$? If some $b_i = 0$? Essentially, by observing that if any b_i is zero the matrix T takes on a block-diagonal form, you should be able to separate the problem of computing the eigenvalues of T into smaller eigenvalue problems with the same symmetric, tridiagonal structure.*

Let $T_k = T_{1:k,1:k}$ denote the kth principal submatrix of T, so that $T = T_n$. Each T_k is symmetric, tridiagonal of size $k \times k$. This suggests we may be able to analyze T through recurrence on the T_k's. In particular, since we are interested in the eigenvalues of T which are the roots of its characteristic polynomial, it makes sense to investigate it.

For each k, define the characteristic polynomial of T_k as:

$$p_k(\lambda) = \det(T_k - \lambda I_k).$$

This is a polynomial of degree k. For example, for $k = 5$, expanding the determinant along the last column for the first matrix and along the last row

for the last matrix we find:

$$\begin{aligned} p_5(\lambda) &= \det \begin{pmatrix} a_1-\lambda & b_2 & & & \\ b_2 & a_2-\lambda & b_3 & & \\ & b_3 & a_3-\lambda & b_4 & \\ & & b_4 & a_4-\lambda & b_5 \\ & & & b_5 & a_5-\lambda \end{pmatrix} \\ &= (a_5-\lambda)\det \begin{pmatrix} a_1-\lambda & b_2 & & & \\ b_2 & a_2-\lambda & b_3 & & \\ & b_3 & a_3-\lambda & b_4 & \\ & & b_4 & a_4-\lambda & \end{pmatrix} \\ &\quad - b_5 \det \begin{pmatrix} a_1-\lambda & b_2 & & & \\ b_2 & a_2-\lambda & b_3 & & \\ & b_3 & a_3-\lambda & b_4 & \\ & & & b_5 & \end{pmatrix} \\ &= (a_5-\lambda)p_4(\lambda) - b_5^2 p_3(\lambda). \end{aligned}$$

Thus, p_5 is easily expressed in terms of the two previous polynomials, p_4 and p_3. This suggests the following three-term recurrence:

$$\begin{aligned} p_1(\lambda) &= a_1-\lambda, \\ p_2(\lambda) &= (a_2-\lambda)(a_1-\lambda) - b_2^2, \\ p_{k+1}(\lambda) &= (a_{k+1}-\lambda)p_k(\lambda) - b_{k+1}^2 p_{k-1}(\lambda), \qquad \text{for } k=2,\dots,n-1. \end{aligned} \tag{5.19}$$

Equivalently, we can define[5]

$$p_0(\lambda) = 1$$

and allow the recurrence (5.19) to run for $k = 1, \dots, n-1$, based on p_0 and p_1.

Question 5.16. *Verify that the recurrence (5.19) for $k = 1$ is valid with the definition $p_0(\lambda) = 1$.*

The following is important:

> *Using the recurrence relation, one can evaluate the characteristic polynomials directly, without ever needing to figure out the coefficients of the polynomials.*

[5] One way to understand the definition of p_0 is that it is the determinant of a 0×0 matrix, which is 1 for the same reason an empty product is 1.

This is important, since we know from the beginning of this chapter that the coefficients of p_n may be badly behaved. On the other hand, evaluating the polynomials using the recurrence is fairly safe.

The following is the main theorem of this section. It tells us how we can use the recurrence relation to determine where the eigenvalues of T are located. Remember we assume $b_2, \ldots, b_n$ are nonzero.

Theorem 5.17 (The Sturm sequence property, Theorem 5.9 in [SM03])**.** *For $\theta \in \mathbb{R}$, consider the Sturm sequence $(p_0(\theta), \ldots, p_n(\theta))$. The number of agreements in sign between consecutive members of the sequence equals the number of eigenvalues of T strictly greater than θ.*

To count agreements in sign in the Sturm sequence, see Figure 5.6.[6] The following code illustrates this theorem; see also Figure 5.3.

```
%% The Sturm sequence property

%% Generate a symmetric, tridiagonal matrix
n = 6;
e = ones(n, 1);
T = spdiags([-e 2*e -e], -1:1, n, n);

% Using the recurrence relation, we can easily evaluate
% the characteristic polynomials of T and its principal
% submatrices for any given theta.
theta = 1.5;
q = zeros(n+1, 1);
q(1) = 1;                   % p_0(x) = 1
q(2) = T(1,1) - theta;      % p_1(x) = a_1 - x
for k = 2 : n
    % p_k(x) = (a_k - x)p_{k-1}(x) - b_k^2 p_{k-2}(x)
    a_k = T(k, k);
    b_k = T(k-1, k);
    q(k+1) = (a_k - theta)*q(k) - b_k^2*q(k-1);
end
stem(0:n, q);
xlim([-1, n+1]);
title(sprintf('Sturm sequence at \\theta = %g', theta));
set(gca, 'XTick', 0:n);
```

The proof of the Sturm sequence property relies crucially on the following theorem. (Remember that, by the spectral theorem, the k roots of p_k are real since T_k is symmetric.) We still assume all b_i's are nonzero.

[6] The rule given in [SM03] is incorrect for Sturm sequences ending with a 0.

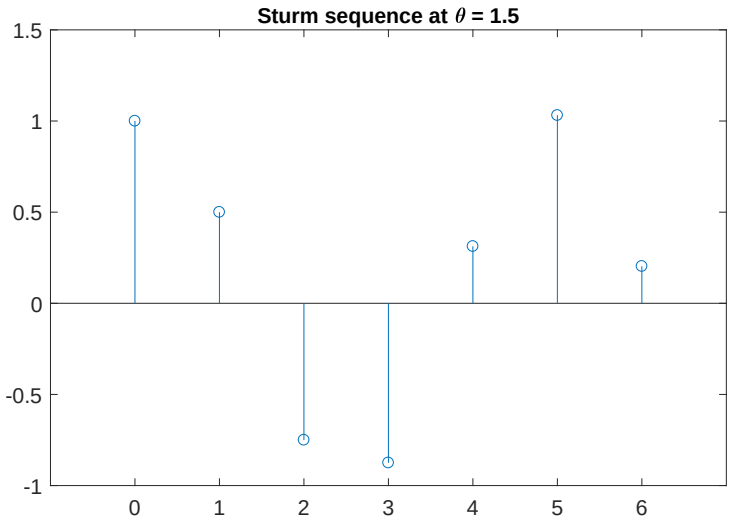


Figure 5.3: Sturm sequence at $\theta = 1.5$ for the 6×6 tridiagonal matrix with 2's on the diagonal and -1's above and below. Its eigenvalues are $0.1981, 0.7530, 1.5550, 2.4450, 3.2470, 3.8019$. Exactly 4 are strictly larger than θ, coinciding with the number of sign agreements in the sequence.

Theorem 5.18 (Cauchy's interlace theorem, Theorem 5.8 in [SM03])**.** *The roots of p_{k-1} are real and distinct, and separate those of p_k for $k = 2, 3, \ldots, n$. That is, (strictly) between any two consecutive roots of p_k (also real and distinct), there is exactly one root of p_{k-1}.*

This theorem is illustrated in Figure 5.4, with a matrix T generated as:

```
%% Generate a specific symmetric, tridiagonal matrix
n = 6;
e = ones(n, 1);
T = spdiags([-e 2*e -e], -1:1, n, n);
```

Question 5.19. *Write your own code to generate Figure 5.4, based on the three-term recurrence. Notice that to evaluate $p_n(\theta)$, you will evaluate $p_0(\theta), p_1(\theta), \ldots, p_n(\theta)$, so that you will be able to plot all polynomials right away. See if you can write your code to evaluate the three-term recurrence at several values of θ simultaneously (using matrix and vector notations).*

The Sturm sequence theorem allows to find any desired eigenvalue of T through bisection. Indeed, define

$$\sharp(x) = \text{ the number of eigenvalues of } T \text{ strictly larger than } x. \tag{5.20}$$

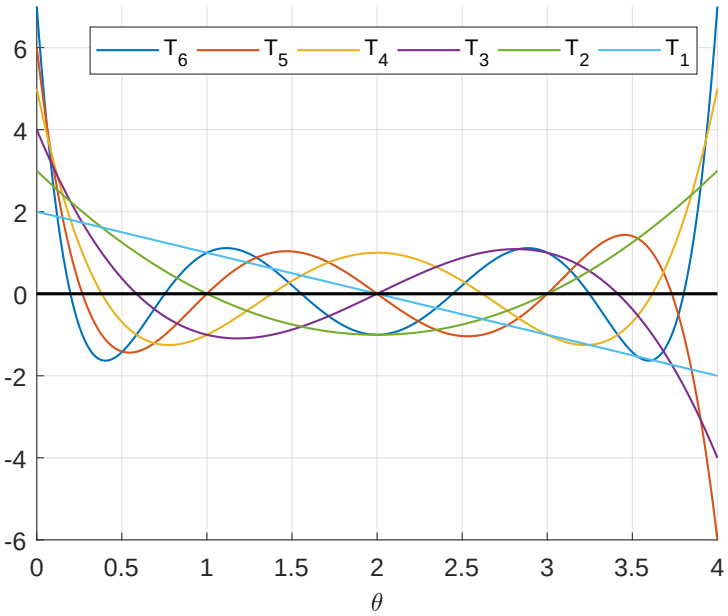


Figure 5.4: Characteristic polynomials of the six principal submatrices of the 6×6 tridiagonal matrix with 2's on the diagonal and -1's above and below. The Cauchy Interlace Theorem explains why the eigenvalues of T_k and T_{k+1} interlace.

This function can be computed easily using Sturm sequences. By Cauchy interlacing, the eigenvalues of T are distinct. As a result, they are separated on the real line as:

$$\lambda_n < \lambda_{n-1} < \cdots < \lambda_3 < \lambda_2 < \lambda_1. \tag{5.21}$$

Say we want to compute λ_k. Start with $a_0 < b_0$ such that $\lambda_k \in (a_0, b_0]$—we will see how to compute such initial bounds later. Consider the middle point $c_0 = \frac{a_0+b_0}{2}$ and compute $\sharp(c_0)$. Two things can happen:

1. Either $\sharp(c_0) \geq k$, indicating $\lambda_1, \ldots, \lambda_k > c_0$—in particular, $\lambda_k > c_0$; or
2. $\sharp(c_0) < k$, indicating $\lambda_k \leq c_0$.

In the first case, we determined that $\lambda_k \in (c_0, b_0]$, while in the second case we found $\lambda_k \in (a_0, c_0]$. In both cases, we found an interval twice smaller than the original interval, still with the guarantee that it contains λ_k. Upon iterating this procedure, we produce an interval of length $|b_0 - a_0|/2^K$ in K iterations, each of which involves a manageable number of operations. This

interval provides both an approximation of the eigenvalue and an error bound for it.

Question 5.20. *How many flops are required for one iteration of this bisection algorithm?*

■

As a side note, observe that $\sharp(a) - \sharp(b)$ gives the number of eigenvalues of T in the interval $(a, b]$.

Proof of Theorem 5.18, see also Theorem 5.8 in [SM03]. We prove Cauchy interlacing by induction. To secure the base case, consider the roots of $p_1(\lambda) = a_1 - \lambda$ and those of $p_2(\lambda) = (a_1 - \lambda)(a_2 - \lambda) - b_2^2$. The unique root of p_1 is a_1, while the two roots of p_2 are:

$$\frac{(a_1 + a_2) \pm \sqrt{(a_1 + a_2)^2 - 4(a_1 a_2 - b_2^2)}}{2} = a_1 + \frac{(a_2 - a_1) \pm \sqrt{(a_2 - a_1)^2 + 4b_2^2}}{2}.$$

Using that $b_2 \neq 0$, we deduce that $\sqrt{(a_2 - a_1)^2 + 4b_2^2} > |a_2 - a_1|$, and from there it is easy to deduce that the two roots of p_2 are real and distinct, and that a_1 (the unique root of p_1) lies strictly between them. Thus, the base case of the induction holds.

We now proceed by induction. The induction hypothesis is that the roots of p_{k-1} interlace those of p_k, and that both have real, distinct roots ($k - 1$ and k respectively). The goal is to infer that p_{k+1} has $k + 1$ real, distinct roots interlaced with those of p_k. We do this in two steps: first, we show that p_{k+1} has at least one root strictly between any two consecutive roots of p_k: this readily accounts for at least $k - 1$ roots. Then, we show that p_{k+1} has a root strictly smaller than all roots of p_k, and (by a similar argument) another root strictly larger than all the roots of p_k. All told, this locates $k + 1$ roots of p_{k+1} as desired, which is all of them.

First, let us show that p_{k+1} admits at least one root between any two consecutive roots of p_k. The key argument is the intermediate value theorem: Let α, β denote two consecutive roots of p_k, with $\alpha < \beta$. Evaluate p_{k+1} at these two points:

$$p_{k+1}(\alpha) = (a_{k+1} - \alpha)p_k(\alpha) - b_{k+1}^2 p_{k-1}(\alpha) = -b_{k+1}^2 p_{k-1}(\alpha),$$
$$p_{k+1}(\beta) = (a_{k+1} - \beta)p_k(\beta) - b_{k+1}^2 p_{k-1}(\beta) = -b_{k+1}^2 p_{k-1}(\beta).$$

By the intermediate value theorem, if $p_{k+1}(\alpha)$ and $p_{k+1}(\beta)$ have opposite signs, then p_{k+1} admits a root in (α, β). According to the above (using that $b_{k+1} \neq 0$), this is the case exactly if $p_{k-1}(\alpha)$ and $p_{k-1}(\beta)$ have opposite signs. This in turn follows from the induction hypothesis: we assume here that the

roots of p_{k-1} and those of p_k interlace, which implies that p_{k-1} has exactly one root in (α, β). Hence, $p_{k-1}(\alpha)$ and $p_{k-1}(\beta)$ have opposite signs as desired. Apply this whole reasoning to each consecutive pair of roots of p_k to locate $k-1$ roots of p_k: we only have two more to locate.

Second, we want to show that p_{k+1} has a root strictly smaller than the smallest root of p_k; let us call the latter γ. The key observation is that all the characteristic polynomials considered here go to positive infinity on the left side, that is, for all r:

$$\lim_{\lambda \to -\infty} p_r(\lambda) = +\infty. \tag{5.22}$$

Indeed, $p_r(\lambda) = \det(T_r - \lambda I_n) = (-\lambda)^r +$ lower order terms.[7] Consider Figure 5.4 for confirmation. We use this observation as follows: consider the sign of

$$p_{k+1}(\gamma) = (a_{k+1} - \gamma)p_k(\gamma) - b_{k+1}^2 p_{k-1}(\gamma) = -b_{k+1}^2 p_{k-1}(\gamma).$$

If this is negative, then p_{k+1} must have a root strictly smaller than γ since it must obey (5.22): $p_{k+1}(\lambda)$ has to become positive eventually as $\lambda \to -\infty$. According to the above, this happens exactly if $p_{k-1}(\gamma)$ is positive (using again that $b_{k+1} \neq 0$). By the induction hypothesis, all the roots of p_{k-1} are strictly larger than γ, and since p_{k-1} itself obeys (5.22), it follows that indeed $p_{k-1}(\gamma) > 0$, as desired. This shows that p_{k+1} has at least one root strictly smaller than γ.

To conclude, we need only show that p_{k+1} has a root strictly larger than the largest root of p_k. The argument is similar to the one above. As only significant difference, we here use the fact that $\lim_{\lambda \to +\infty} p_{k-1}(\lambda)$ and $\lim_{\lambda \to +\infty} p_{k+1}(\lambda)$ are both infinite *of the same sign.* □

Before we get into the proof of the Sturm sequence property, let us make three observations about how zeros may appear in such sequences:

1. The first entry of the sequence is always + (in particular, it is not 0).

2. There can never be two subsequent zeros. Indeed, assume for contradiction that $p_k(\theta) = p_{k-1}(\theta) = 0$. Then, the recurrence implies $0 = -b_k^2 p_{k-2}(\theta)$. Thus, $p_{k-2}(\theta) = 0$ (under our assumption that $b_k \neq 0$.) Applying this same argument to $p_{k-1}(\theta) = p_{k-2}(\theta) = 0$ implies $p_{k-3}(\theta) = 0$, etc. Eventually, we conclude $p_0(\theta) = 0$, which is a contradiction since $p_0(\theta) = 1$ for all θ.

[7] You can also see it from the recurrence relation (5.19), which shows the sign of the highest order term changes at every step.

3. When a zero occurs before the end of the sequence, it is followed by the sign opposite the sign that preceded it. Specifically, patterns $(+,0,-)$ and $(-,0,+)$ can occur, but patterns $(+,0,+)$ and $(-,0,-)$ cannot. Indeed: if $p_k(\theta)=0$ with $k<n$, then $p_{k-1}(\theta)$ and $p_{k+1}(\theta)$ are nonzero because of the previous point. Furthermore, the recurrence states $p_{k+1}(\theta)=-b_{k+1}^2 p_{k-1}(\theta)$, hence they have opposite signs.

We now give a proof of the main theorem. The proof differs somewhat from [SM03, Theorem 5.9]. Specifically, we handle the case of zeros in the Sturm sequence.

Proof of Theorem 5.17. The Sturm sequence property is a theorem about two quantities. Let us give them a name:

$$\begin{aligned} s_k(\theta) &= \text{number of sign agreements in } (p_0(\theta),\dots,p_k(\theta)), \text{ and} \\ g_k(\theta) &= \text{number of roots of } p_k \text{ strictly larger than } \theta. \end{aligned}$$

Our goal is to show that $s_n(\theta)=g_n(\theta)$. By induction, we prove that $s_k(\theta)=g_k(\theta)$ for all k, which implies the result. The first step is to secure the base case, $k=1$. By definition,

$$s_1(\theta)=\text{number of sign agreements in } (1,a_1-\theta)=\begin{cases}1 & \text{if } a_1-\theta>0,\\ 0 & \text{otherwise.}\end{cases}$$

Furthermore, since the unique root of p_1 is a_1,

$$g_1(\theta)=\text{number of roots of } p_1 \text{ strictly larger than } \theta=\begin{cases}1 & \text{if } a_1>\theta,\\ 0 & \text{otherwise.}\end{cases}$$

Clearly, $g_1(\theta)=s_1(\theta)$ as desired. Now, under the induction hypothesis $s_{k-1}(\theta)=g_{k-1}(\theta)$, let us prove that $s_k(\theta)=g_k(\theta)$. In order to do so, notice that $s_k(\theta)$ and $g_k(\theta)$ can be expressed in terms of $s_{k-1}(\theta)$ and $g_{k-1}(\theta)$. Indeed,

$$s_k(\theta)=\begin{cases}s_{k-1}(\theta)+1 & \text{if } (p_{k-1}(\theta),p_k(\theta)) \text{ agree in sign,}\\ s_{k-1}(\theta) & \text{otherwise.}\end{cases}$$

Similarly, owing to the Cauchy interlacing theorem, for any θ, the difference $g_k(\theta)-g_{k-1}(\theta)$ can be either 0 or 1. Specifically,

$$g_k(\theta)=\begin{cases}g_{k-1}(\theta)+1 & \text{if, compared to } p_{k-1},\ p_k \text{ has one more root } >\theta,\\ g_{k-1}(\theta) & \text{otherwise.}\end{cases}$$

Thus, using the induction hypothesis $s_{k-1}(\theta) = g_{k-1}(\theta)$, in order to show that $s_k(\theta) = g_k(\theta)$, we only need to verify that the following conditions are in fact equivalent:

1. $(p_{k-1}(\theta), p_k(\theta))$ agree in sign;

2. Compared to p_{k-1}, p_k has one more root strictly larger than θ.

This is best checked on a drawing: see Figure 5.5. There are four cases to verify, using Cauchy interlacing and the fact that $\lim_{\lambda\to-\infty} p_r(\theta) = +\infty$ for $r = k-1, k$.

1. θ is a root of p_k: then the sequence is $(p_{k-1}(\theta), 0)$. There is no sign agreement, and indeed p_k and p_{k-1} have the same number of roots strictly larger than θ.

2. $\theta <$ the smallest root of p_k: then both $p_{k-1}(\theta)$ and $p_k(\theta)$ are positive. We have a sign agreement, and indeed p_k has k roots strictly larger than θ, which is one more than p_{k-1}.

3. $\theta >$ the largest root of p_k: then $p_{k-1}(\theta)$ and $p_k(\theta)$ (nonzero) have opposite sign. We have no sign agreement, and indeed p_k and p_{k-1} have the same number of roots strictly larger than θ, namely, zero.

4. $\theta \in (\alpha, \beta)$, where α, β are two consecutive roots of p_k. By Cauchy interlacing, p_{k-1} has exactly one root $\gamma \in (\alpha, \beta)$. There are three cases:

   (a) $\theta \in (\alpha, \gamma)$: there is no sign agreement since $p_{k-1}(\theta)$ and $p_k(\theta)$ (both nonzero) have opposite sign (check this by following the sign patterns on a drawing). And indeed, since θ passed one more root of p_k than it passed roots of p_{k-1}, there are an equal number of roots of p_k and p_{k-1} strictly larger than θ.

   (b) $\theta = \gamma$: the sequence is $(0, p_k(\theta))$. There is a sign agreement, and indeed θ passed as many roots of p_k as it passed roots of p_{k-1}, so that p_k has one more strictly larger than θ;

   (c) $\theta \in (\gamma, \beta)$: there is sign agreement (for the same reason that there was no sign agreement in the first case), and there is one more root of p_k to the right of θ than there are roots of p_{k-1}.

Hence, $s_k(\theta) = g_k(\theta)$, and by induction $s_n(\theta) = g_n(\theta)$, as desired. □

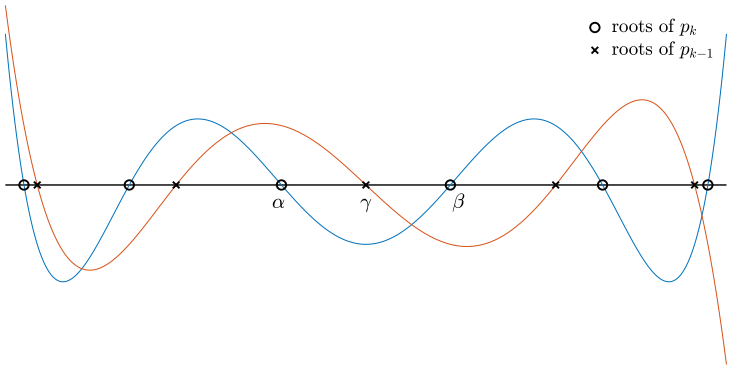


Figure 5.5: The Cauchy interlacing theorem completely characterizes the sign patterns of p_k and p_{k-1}, which allows to relate the sign agreements in the Sturm sequence at θ to the number of roots of p_k strictly larger than θ.

5.5 Locating eigenvalues: Gerschgorin

We cover two simple approaches to find rough bounds on the location of the eigenvalues of matrices in the complex plane. For symmetric matrices, these yield real intervals which contain the eigenvalues: such intervals can be used to start a bisection based on Sturm sequences.

Technique 1. The first technique is based on subordinate norms. If (λ, x) is an eigenpair of $A \in \mathbb{C}^{n\times n}$, then $\lambda x = Ax$, which implies (reading from the middle):

$$|\lambda|\|x\| = \|\lambda x\| = \|Ax\| \leq \|A\|\|x\|, \tag{5.23}$$

for any choice of vector norm $\|\cdot\|$. Since x is an eigenvector, it is nonzero and we get

$$|\lambda| \leq \|A\|, \tag{5.24}$$

where $\|A\|$ is the norm of A, subordinate to the vector norm $\|\cdot\|$. For symmetric matrices, this means each eigenvalue is in the interval $[-\|A\|, +\|A\|]$. For this observation to be practical, it only remains to pick a vector norm whose subordinate matrix norm is straightforward to compute. A poor choice is the 2-norm (since the subordinate norm is the largest singular value: this is as hard to compute as an eigenvalue); good choices are the 1-norm and ∞-norm:

Any eigenvalue λ of A obeys $|\lambda| \leq \|A\|_1$ and $|\lambda| \leq \|A\|_\infty$.

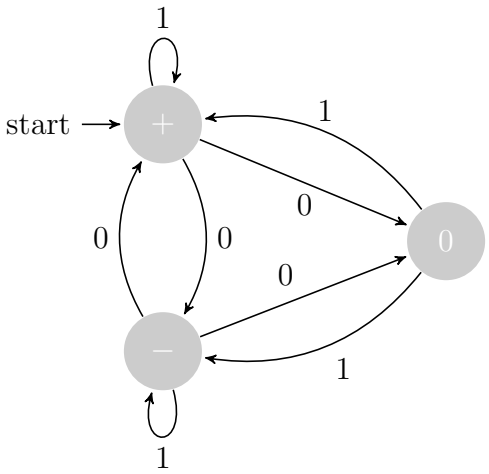


Figure 5.6: Diagram to count sign agreements in a Sturm sequence. For example, the sequence $(+,+,-,0,+,0,-,-,0)$ has 4 sign agreements: $(+,+),(0,+),(0,-),(-,-)$. As argued in the text, there cannot be two consecutive zeros. Furthermore, the patterns $(+,0,+)$ and $(-,0,-)$ cannot occur. This suggests one way to handle zeros: replace them with the sign opposite to the previous entry. With the above example, we get $(+,+,-,+,+,-,-,-,+)$, which indeed still has 4 sign agreements.

These bounds are indeed trivial to compute.

Question 5.21. *Prove that* $\|A\|_\infty = \max\limits_{i=1,\dots,n} \sum\limits_{j=1}^{n} |a_{ij}|$ *and* $\|A\|_1 = \|A^T\|_\infty$.

■

Thus, we can start the Sturm bisection with the interval $[-\|A\|_1, \|A\|_1]$ or $[-\|A\|_\infty, \|A\|_\infty]$.[8]

Technique 2. The above strategy determines one large disk in the complex plane which contains all eigenvalues. If the eigenvalues are spread out, this can only give a very rough estimate of their location. A more refined approach consists in determining a collection of (usually) smaller disks whose union contains all eigenvalues. These disks are called *Gerschgorin disks*. The code below produces Figure 5.7; then we will see how they are constructed.

[8]The Sturm bisection technically works with intervals of the form $(a,b]$, not $[a,b]$. Hence, if the smallest eigenvalue is targeted, one should check whether $-\|A\|_1$ or $-\|A\|_\infty$ is a root of p_n, simply by evaluating it there, or one can make the interval slightly larger.

```
%% Locating eigenvalues
A = [7 2 0 ; -1 8 1 ; 2 2 0];
e = eig(A);
plot(real(e), imag(e), '.', 'MarkerSize', 25);
xlabel('Real');
ylabel('Imaginary');
title('Eigenvalues of A');
xlim([-15, 15]);
ylim([-15, 15]);
axis equal;
hold on;

%%
% subordinate norms
circles(0, 0, norm(A, 1),   'FaceAlpha', .1);
circles(0, 0, norm(A, inf), 'FaceAlpha', .1);
% The code for 'circles' is here:
% https://www.mathworks.com/matlabcentral/ ...
% fileexchange/45952-circle-plotter

%%
% Gerschgorin disks
for k = 1 : size(A, 1)
    radius = sum(abs(A(k, [(1:k-1), (k+1:end)])));
    circles(real(A(k, k)), imag(A(k, k)), radius, ...
       'FaceAlpha', .1);
end

hold off;
```

Let's go through the construction, which will amount to a proof of the theorem below. Pick any eigenpair (λ, x) of A. Let k be such that $|x_i| \leq |x_k|$ for all i. Note that $x_k \neq 0$ since $x \neq 0$. Since $\lambda x = Ax$, we have in particular:

$$\lambda x_k = (Ax)_k = \sum_{i=1}^{n} a_{ki} x_i = a_{kk} x_k + \sum_{i \neq k} a_{ki} x_i. \tag{5.25}$$

Reorganizing:

$$(\lambda - a_{kk}) x_k = \sum_{i \neq k} a_{ki} x_i. \tag{5.26}$$

Using the triangular inequality and the defining property of k,

$$|\lambda - a_{kk}| \cdot |x_k| = \left| \sum_{i \neq k} a_{ki} x_i \right| \leq \sum_{i \neq k} |a_{ki}| \cdot |x_i| \leq |x_k| \sum_{i \neq k} |a_{ki}|. \tag{5.27}$$

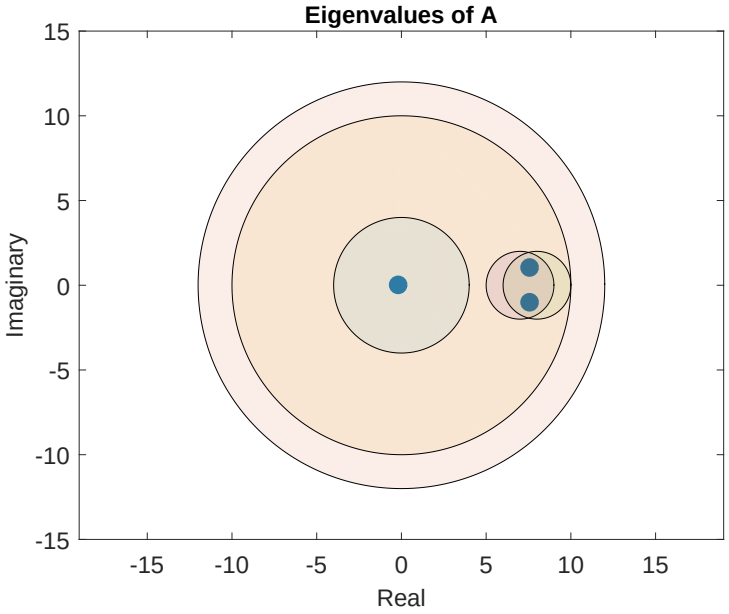


Figure 5.7: Comparison of subordinate-norm disks and Gerschgorin disks in the complex plane to locate the eigenvalues of $A = \begin{bmatrix} 7 & 2 & 0 \\ -1 & 8 & 1 \\ 2 & 2 & 0 \end{bmatrix}$ (blue dots). The largest disk has radius $\|A\|_1 = 12$. The second largest has radius $\|A\|_\infty = 10$. The latter two are centered at the origin. The three smaller disks are the Gerschgorin disks, centered at the diagonal entries of A.

Finally, using $x_k \neq 0$ we have:

$$|\lambda - a_{kk}| \leq \sum_{i \neq k} |a_{ki}|. \tag{5.28}$$

This indeed guarantees any eigenvalue λ is in at least one of n possible disks (corresponding to all possible values of k). Since we do not know a priori which eigenvalue is in which disk, the safest statement we can make at this point is that all eigenvalues lie in the union of all n disks.

Theorem 5.22 (Gerschgorin disks, Theorem 5.4 in [SM03])**.** *Consider a matrix $A \in \mathbb{C}^{n \times n}$. All eigenvalues of A lie in the region $D = \cup_{k=1}^{n} D_k$, where $D_k = \{z \in \mathbb{C} : |z - a_{kk}| \leq \sum_{i \neq k} |a_{ki}|\}$.*

Question 5.23. *How do you use Gerschgorin disks to produce an interval which contains all eigenvalues of a symmetric matrix A?*

Question 5.24. *A symmetric matrix A is* diagonally dominant *if, for all k, $a_{kk} \geq \sum_{i \neq k} |a_{ki}|$. Show that such matrices are positive semidefinite.)*

5.6 Tridiagonalizing matrices: Householder

The Sturm sequence property only applies to tridiagonal matrices. In this section, we show how to reduce any symmetric matrix of size n to a tridiagonal, symmetric matrix with the same eigenvalues, in $O(n^3)$ flops. This tridiagonalization algorithm is useful in a number of other contexts too (notably for RQI). See Lecture 26 in [TBI97], and a bit of Lecture 10 for Householder reflectors.

Problem 5.25. *Given a symmetric matrix A of size $n \times n$, find a symmetric matrix T which is tridiagonal and has the same eigenvalues as A.*

A first step is to identify which operations one can apply to a matrix without affecting its eigenvalues. The answer is *similarity transforms*. That is, for any invertible matrix M of size n, it holds that $M^{-1}AM$ and A have the same eigenvalues. Indeed, consider the characteristic polynomial of $M^{-1}AM$:

$$\begin{aligned} p_{M^{-1}AM}(\lambda) &= \det\left(M^{-1}AM - \lambda I_n\right) \\ &= \det\left(M^{-1}AM - \lambda M^{-1}I_n M\right) \\ &= \det\left(M^{-1}(A - \lambda I_n)M\right) \\ (\text{using } \det(CD) = \det(C)\det(D)) \quad &= \det(M^{-1})\det(A - \lambda I_n)\det(M) \\ (\text{again}) \quad &= p_A(\lambda)\det(M^{-1}M) \\ &= p_A(\lambda), \end{aligned}$$

where $p_A(\lambda)$ is the characteristic polynomial of A. We find that A and $M^{-1}AM$ have the same characteristic polynomial, hence they have the same eigenvalues.

A second step is to restrict our attention to similarity transforms that preserve the symmetry of A. Thus, we consider only invertible matrices M such that $M^{-1}AM$ is symmetric. In other words, using $A = A^T$:

$$M^{-1}AM = (M^{-1}AM)^T = M^T A^T (M^{-1})^T = M^T A (M^{-1})^T.$$

This is true in particular if $M^{-1} = M^T$, that is, if M is *orthogonal*. Overall, we decide to restrict our attention to orthogonal transformations of A:

Problem 5.26. *Given a symmetric matrix A of size $n \times n$, find an orthogonal matrix Q of size $n \times n$ such that $T = Q^T AQ$ is tridiagonal.*

It is clear that T is symmetric, and as we just argued it has the same eigenvalues as A. How do we construct such a matrix Q?[9] Consider a 5×5

[9] We give a brief description here: see the textbook for a full description. In particular, see the textbook for a discussion of flop count, and for practical implementation considerations.

matrix for example:

$$A = \begin{bmatrix} \times & \times & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \times & \times & \times & \mathbf{\times} & \mathbf{\times} \\ \mathbf{\times} & \times & \times & \times & \mathbf{\times} \\ \mathbf{\times} & \mathbf{\times} & \times & \times & \times \\ \mathbf{\times} & \mathbf{\times} & \mathbf{\times} & \times & \times \end{bmatrix}.$$

The goal is to transform all the bold entries into zeros. Let us focus on the first column first: we want to find an orthogonal matrix Q_1 such that $Q_1^T A$ will have zeros in the indicated entries of the first column. Here is a crucial point:

> *Applying Q_1^T to A on the left will replace each one of the rows of A with a linear combination of the rows of A. We are about to design Q_1^T such that zeros appear in the first column. Since we will then have to apply Q_1 on the right, it is crucial that right-multiplication with Q_1 should leave the first* column *untouched, as otherwise it might scramble our hard-earned zeros. Thus, we need Q_1^T applied on the left to leave the first* row *untouched.*

In other words, we need Q_1 to assume this form:

$$Q_1^T = \begin{bmatrix} 1 & 0 \\ 0 & H_1 \end{bmatrix},$$

where $H_1 \in \mathbb{R}^{(n-1)\times(n-1)}$ is orthogonal (so that Q_1 is orthogonal: think about it). Indeed, applying Q_1^T on the left has no effect on the first row, and applying Q_1 on the right has no effect on the first column. With a proper choice of H_1, we will have

$$Q_1^T A = \begin{bmatrix} \times & \times & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \times & \times & \times & \mathbf{\times} & \mathbf{\times} \\ 0 & \times & \times & \times & \mathbf{\times} \\ 0 & \mathbf{\times} & \times & \times & \times \\ 0 & \mathbf{\times} & \mathbf{\times} & \times & \times \end{bmatrix},$$

and when we apply Q_1 on the right, the zeros will endure. Furthermore, since $Q_1^T A Q_1$ is symmetric, and since we know it has zeros in the first column, it must also have zeros in the first row:

$$Q_1^T A Q_1 = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & \mathbf{\times} & \mathbf{\times} \\ 0 & \times & \times & \times & \mathbf{\times} \\ 0 & \mathbf{\times} & \times & \times & \times \\ 0 & \mathbf{\times} & \mathbf{\times} & \times & \times \end{bmatrix}.$$

From here, it is only a matter of iterating the same idea. Specifically, let us design Q_2^T such that applying it to the left will make zeros appear in the right places in the second column. Upon doing this, we must make sure not to affect the first two rows, so that when we apply Q_2 to the right, our work in columns 1 and 2 will be unaffected. Specifically, build an orthogonal matrix H_2 of size $n-2$ such that

$$Q_2^T Q_1^T A Q_1 = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix},$$

where

$$Q_2^T = \begin{bmatrix} I_2 & 0 \\ 0 & H_2 \end{bmatrix}.$$

By the same symmetry argument, applying Q_2 on the right yields zeros in the second row, preserving the other zeros produced so far:

$$Q_2^T Q_1^T A Q_1 Q_2 = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix}.$$

Similarly, build H_3 orthogonal of size $n-3$ such that

$$\underbrace{Q_3^T Q_2^T Q_1^T}_{Q^T} A \underbrace{Q_1 Q_2 Q_3}_{Q} = \underbrace{\begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}}_{T},$$

where

$$Q_3^T = \begin{bmatrix} I_3 & 0 \\ 0 & H_3 \end{bmatrix}.$$

In general, define

$$Q = Q_1 Q_2 \cdots Q_{n-2}.$$

This matrix is indeed orthogonal because a product of orthogonal matrices is orthogonal. Furthermore, as announced, we have

$$Q^T A Q = T$$

where T is tridiagonal. Equivalently, we can also write

$$A = QTQ^T,$$

which is sometimes more convenient. If only the eigenvalues of A are of interest, there is no need to build the matrix Q explicitly: it is only necessary to produce T. If the eigenvectors are desirable, there are computationally efficient ways of building them without direct construction of Q: see below for a few words on that topic, and the textbook for details.

At this point, the only missing piece is: how do we construct the orthogonal transformations $H_1, H_2, \ldots, H_{n-2}$? One practical way is to use *Householder reflectors*: see Lecture 10 in [TBI97]. Specifically, let us consider what such a matrix H must satisfy:

1. $H \in \mathbb{R}^{k\times k}$ must be orthogonal, and
2. For a given vector $x \in \mathbb{R}^k$, we must have $Hx = \begin{bmatrix} \times \\ 0 \\ \vdots \\ 0 \end{bmatrix}$.

In our setting, the vector x that drives the construction of H is extracted from the column currently under inspection. A simple observation is that, since H is orthogonal, it has no effect on the 2-norm of a vector. Hence, $\|Hx\| = \|x\|$. As a result, the second requirement can be written as

$$Hx = \pm\|x\|e_1, \tag{5.29}$$

where e_1 is the first canonical vector in $\mathbb{R}^k$, that is, $e_1^T = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}$. Either sign is good: we will commit later. How do we construct a matrix H satisfying these requirements? One good way is by reflection.[10] Specifically, consider

$$u = \frac{x - Hx}{\|x - Hx\|}. \tag{5.30}$$

[10] See Figure 10.1 in [TBI97], where their F is our H and their normalized v is our u.

This unit-norm vector defines a plane normal to it. We will reflect x across that plane. To this end, consider the orthogonal projection of x to the span of u: it is $(u^T x)u$. Hence, the orthogonal projection of x to the plane itself is

$$Px = x - (u^T x)u.$$

We can reflect x across the plane by moving from x to the plane, orthogonally, then traveling the exact same distance once more (along the same direction). Thus,

$$Hx = x + 2(Px - x) = 2Px - x = x - 2(u^T x)u = (I_k - 2uu^T)x.$$

In other words, a good candidate is

$$H = I_k - 2uu^T.$$

Since $\|u\| = 1$, it is clear that H is orthogonal. Indeed,

$$H^T H = (I_k - 2uu^T)^T(I_k - 2uu^T) = I_k - 4uu^T + 4u(u^T u)u^T = I_k,$$

so that $H^T = H^{-1}$. If we can find an efficient way to compute u, then we have an efficient way to find H. To this end, combine (5.29) and (5.30):

$$u \propto x \mp \|x\|e_1,$$

where by $\propto$ we mean that the vector on the left hand side is proportional to the vector on the right hand side. The vector on the right hand side is readily computed, and furthermore we know that u has unit norm, so that it is clear how to compute u in practice: compute the right hand side, then normalize. How do we pick the sign? Floating-point arithmetic considerations suggest to pick the sign such that $\text{sign}(x_1)$ and $\text{sign}(\mp\|x\|) = \text{sign}(\mp)$ agree (think about it). Overall, given $x \in \mathbb{R}^k$, we can compute u as follows:

1. $u \leftarrow x + \text{sign}(x_1)\|x\|e_1$,
2. $u \leftarrow \frac{u}{\|u\|}$.

This costs $\sim 3k$ flops. The corresponding reflector is $H = I_k - 2uu^T$. In practice, we would not compute H explicitly: that would be quite expensive. Instead, observe that applying H to a matrix can be done efficiently by exploiting its structure, namely:

$$HM = M - 2u(u^T M).$$

Computing the vector $u^T M$ costs $\sim 2k^2$ flops. With $u^T M$ computed, computing HM costs another $\sim 2k^2$ flops. This is much cheaper than if we form H explicitly ($\sim k^2$), then compute a matrix-matrix product, at $\sim 2k^3$ flops. Of course, we can also apply H on the right, as in MH, for $\sim 4k^2$ as well.

To compute $2uv^T$, first multiply u by 2 for k flops, then do the vector product for k^2 flops (as opposed to doing the vector product first, then multiplying each entry by 2, which costs $2k^2$ flops.)

Question 5.27. *Using the above considerations, how many flops do you need to compute the matrix T, given the matrix A? You should find that, without exploiting the symmetry of A, you can do this in $\sim \frac{10}{3}n^3$ flops.*

■

Question 5.28. *The vectors $u_1, \ldots, u_{n-2}$ produced along the way (to design the reflectors $H_1, \ldots, H_{n-2}$) can be saved in case it becomes necessary to apply Q to a vector later on. How many flops does it take to apply Q to a vector $y \in \mathbb{R}^n$ in that scenario? How does that compare to the cost of a matrix-vector product Qy as if we had Q available as a matrix directly?*

Question 5.29. *What happens if we apply the exact same algorithm to a non-symmetric matrix A?*

■

Remark 5.30. *Householder reflectors can also be used to compute QR factorizations: this is what Matlab uses for* `qr`*; see Lecture 10 in [TBI97]. Furthermore, Householder tridiagonalization is the initial step for the so-called* QR algorithm *for eigenvalue computations: see Lectures 28, 29 in [TBI97]. Subsequent operations in that algorithm perform QR factorizations iteratively (hence the name QR algorithm: it is not an algorithm to compute QR factorizations; it is an algorithm to compute eigenvalue decompositions, using QR factorizations). This is quite close to the algorithm Matlab uses for* `eig`*.*

Chapter 6

Polynomial interpolation

This chapter is based on Chapter 6 in [SM03] and focuses on the following problem.

Problem 6.1. *Given $x_0, \ldots, x_n \in \mathbb{R}$ distinct and $y_0, \ldots, y_n \in \mathbb{R}$, find a polynomial p of degree at most n such that $p(x_i) = y_i$ for $i = 0, \ldots, n$.*

It is clear that the polynomial should be allowed to have degree up to n in general, since there are $n+1$ constraints: we need $n+1$ degrees of freedom. It so happens that degree n is always sufficient as well, as will become clear.

It is good to clarify a few points of vocabulary here:

- Polynomial interpolation is the problem spelled out above;
- Polynomial regression is the problem of finding a polynomial such that $p(x_i) \approx y_i$ (where approximation is best in some sense): this usually allows to pick a lower-degree polynomial, and makes more sense if the data is noisy;
- Polynomial approximation is the problem of approximating a function f with a polynomial; it comes in at least two flavors:
  - Taylor polynomials approximate f and its derivatives at a specific point;
  - Minimax and least-squares polynomials (which we discuss in later lectures) approximate f over a whole interval.
- Polynomial extrapolation consists in using the obtained polynomial at points x outside the range for which one has data: this is usually risky business.

The key insight to solve the interpolation problem is to notice that

$$\mathcal{P}_n = \{ \text{ polynomials of degree at most } n \}$$

is a linear subspace of dimension $n+1$ (see also later in this section). Hence, upon choosing a basis for it, any element in $\mathcal{P}_n$ can be identified with $n+1$ coefficients. It is all about picking a convenient basis.

The obvious choice is the so-called monomial basis:

$$1, x, x^2, \ldots, x^n.$$

Then, a candidate solution $p \in \mathcal{P}_n$ can be written as

$$p(x) = \sum_{k=0}^{n} a_k x^k$$

with coefficients $a_0, \ldots, a_n$ to be determined. Each equation is of the form:

$$y_i = p(x_i) = \sum_{k=0}^{n} a_k x_i^k.$$

This is a linear equation in the coefficients. Collecting them for $i = 0, \ldots, n$ yields the system:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}}_{\text{Vandermonde matrix}} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{6.1}$$

Thus, it is sufficient to solve this linear system involving the Vandermonde matrix to obtain the coefficients of the solution p in the monomial basis. The Vandermonde matrix is invertible provided the x_i's are distinct[1] (as we assume throughout), but it is unfortunately ill-conditioned in general.[2]

This means that for most choices of interpolation points x_i we cannot hope to solve this linear system with any reasonable accuracy for n beyond a small threshold. This is illustrated for equispaced interpolation points and Chebyshev interpolation points (to be discussed later) in Figure 6.1.

[1] We do not prove this explicitly, though it is implied implicitly by results to follow.

[2] One notable exception is for the complex case where $x_j = e^{2\pi i \frac{j}{n+1}}$ (the $n+1$ complex roots of unity): then the Vandermonde matrix is in fact the Fourier matrix, unitary up to a scaling. Its condition number is 1!

```
% Condition number of Vandermonde matrix
a = 0; b = 1;
kk = 20;
cond1 = zeros(kk, 1);
cond2 = zeros(kk, 1);
for k = 1:kk
    x1 = linspace(a, b, k+1); % equispaced
    x2 = (a+b)/2 + (b-a)/2 * cos( (2*(0:k) + 1)*pi ./ ...
        (2*k+2) ); % Chebyshev
    cond1(k) = cond(vander(x1));
    cond2(k) = cond(vander(x2));
end
semilogy(1:kk, cond1, '.-', 1:kk, cond2, '.-');
```

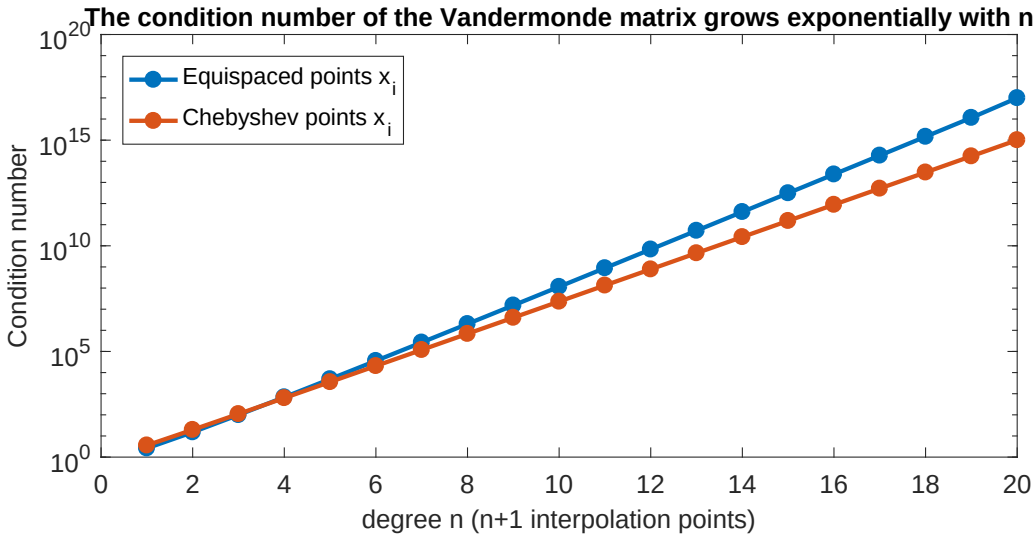


Figure 6.1: For equispaced and Chebyshev interpolation points, the Vandermonde matrix has exponentially growing condition number. Beyond degree 20 or so, one cannot hope to compute the coefficients of the interpolation polynomial in the monomial basis with any reasonable accuracy in double precision arithmetic. This does not mean, however, that the interpolation polynomial cannot be evaluated accurately: only that methods based on computing the coefficients in the monomial basis first are liable to incur a large error.

Sets of polynomials as vector spaces

In your linear algebra course, you may have only considered vectors and subspaces in $\mathbb{R}^n$. For that important particular case, the properties we use over and over again are:

1. The zero vector belongs to $\mathbb{R}^n$,
2. Adding two vectors from $\mathbb{R}^n$ yields a vector of $\mathbb{R}^n$, and
3. Multiplying a vector of $\mathbb{R}^n$ by a scalar yields a vector of $\mathbb{R}^n$.

In fact, these are the *only* properties of $\mathbb{R}^n$ that we really need in order to develop the core concepts and results from your linear algebra course, including the notions of linear independence, bases, dimension, subspaces, changes of coordinates, linear transformations, etc.

Now, consider the following statements about $\mathcal{P}_n$, the set of polynomials of degree at most n, where $n \geq 0$:

1. The zero polynomial is a polynomial of degree at most n,
2. Adding two polynomials of degree at most n yields a polynomial of degree at most n, and
3. Multiplying a polynomial of degree at most n by a scalar yields a polynomial of degree at most n.

Thus, $\mathcal{P}_n$ satisfies all the same properties as $\mathbb{R}^n$ insofar as we are concerned for linear algebra purposes. This is why we say $\mathcal{P}_n$ is a vector space (or a linear space, or a linear subspace, or simply a subspace), and why we have notions of linearly independent polynomials, bases for $\mathcal{P}_n$, dimension of $\mathcal{P}_n$ and of its subspaces, changes of coordinates for polynomials, linear transformations on polynomials, etc.

More advanced concepts of your linear algebra course required an additional tool on $\mathbb{R}^n$: the notion of an inner product (or dot product). This allowed us to define such things as orthogonal projections and orthonormal bases. We will see that inner products can be defined over $\mathcal{P}_n$ as well, so that we can define orthonormal bases of polynomials.

This is an abstract concept: take some time to let it sink in.

6.1 Lagrange interpolation, the Lagrange way

So far, we made only one arbitrary choice: working with the monomial basis. Clearly, that won't do. Thus, we must aim to find a better basis. Here,

"better" means that the matrix appearing in the linear system should have better condition number that the Vandermonde matrix. In doing so, we might as well be greedy and aim for the best conditioned matrix of all: the identity matrix.

We are looking for a basis of $\mathcal{P}_n$ made of $n+1$ polynomials of degree at most n,

$$L_0(x), \ldots, L_n(x), \tag{6.2}$$

such that the interpolation problem reduces to a linear system with an identity matrix. Using this (for now unknown) basis, the solution p can be written as

$$p(x) = \sum_{k=0}^{n} a_k L_k(x).$$

Enforcing $p(x_i) = y_i$ for each i, we get the following linear system of $n+1$ equations in $n+1$ unknowns:

$$\begin{bmatrix} L_0(x_0) & L_1(x_0) & L_2(x_0) & \cdots & L_n(x_0) \\ L_0(x_1) & L_1(x_1) & L_2(x_1) & \cdots & L_n(x_1) \\ \vdots & \vdots & \vdots & & \vdots \\ L_0(x_n) & L_1(x_n) & L_2(x_n) & \cdots & L_n(x_n) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{6.3}$$

For the matrix to be identity, we need each L_k to satisfy the following (consider each column separately):

$$L_k(x_k) = 1, \text{ and } L_k(x_{i\neq k}) = 0. \tag{6.4}$$

The latter specifies n roots for each L_k, only leaving a scaling indeterminacy; that scaling is determined by the former condition. There is only one possible choice:

$$L_k(x) = \frac{\prod_{i\neq k}(x - x_i)}{\prod_{i\neq k}(x_k - x_i)}. \tag{6.5}$$

The numerator is determined by the requirement to have roots $x_{i\neq k}$, and the denominator is determined by fixing the scaling $L_k(x_k) = 1$. These polynomials are called the *Lagrange polynomials* for the given x_i's. Each L_k has degree n.

Question 6.2. *Show the Lagrange polynomials form a basis of $\mathcal{P}_n$.*

■

Since the matrix is identity, the system of equations is easily solved: $a_k = y_k$ for each k, and the interpolation polynomial is:

$$p(x) = \sum_{k=0}^{n} y_k L_k(x). \tag{6.6}$$

It is clear by construction that $p(x_i) = y_i$.

It is important to note here that the polynomials L_k are *not* meant to be expanded into the monomial basis. Consider (6.6) as the final answer. To evaluate $p(x)$, one approach is to simply plug x into the formula (6.5) for each L_k, then form the linear combination (6.6). This requires $O(n^2)$ work for each x. A cheaper and numerically better approach is to use the barycentric formula. See for example https://en.wikipedia.org/wiki/Lagrange_polynomial#Barycentric_form. This takes $O(n^2)$ preprocessing work (to be done once, could be saved to disk), then $O(n)$ for each point: as much work as if one had an expression for p in the monomial basis.

Question 6.3. *Implement a function* `f = interp_lagrange_bary(x, y, t)` *which returns the evaluation of the interpolation polynomial through data* (x, y) *at points in* `t`*; your code should use the barycentric formula mentioned above.*

■

The following program allows to visualize the Lagrange polynomials for equispaced points on an interval, see Figure 6.2.

```
% Lagrange polynomial basis
n = 5;
x = linspace(-1, 1, n+1);
I = eye(n+1);
t = linspace(min(x), max(x), 251);
for k = 1 : n+1

    subplot(2, 3, k);

    % Evaluate the Lagrange polynomials by interpolating
    % the standard basis vectors.
    ek = I(k, :);
    plot(t, interp_lagrange_bary(x, ek, t), 'LineWidth', 1.5);

    hold all;
    stem(x, ek, '.', 'MarkerSize', 15, 'LineWidth', 1.5);
```

```
    hold off;

    ylim([-1, 1.5]);
    set(gca, 'YTick', [-1, 0, 1]);
    set(gca, 'XTick', [-1, 0, 1]);

end
```

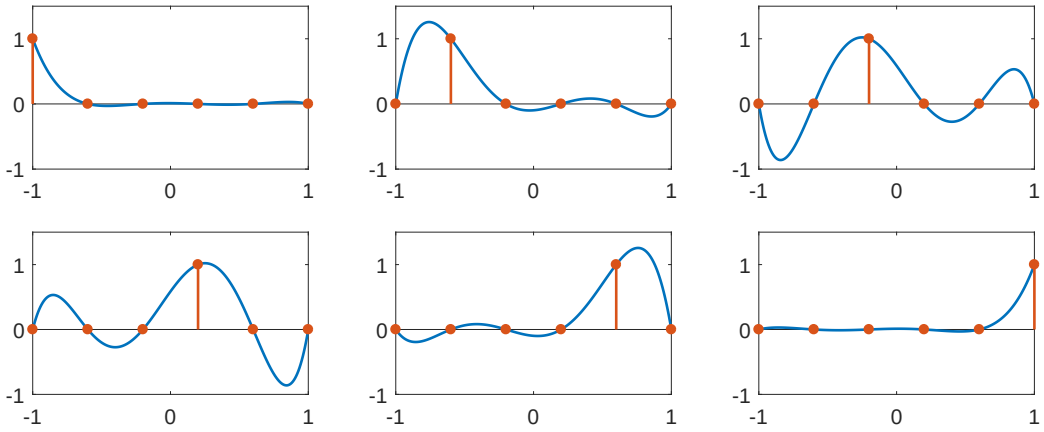


Figure 6.2: Lagrange polynomials for 6 equispaced points in $[-1, 1]$.

It is clear from above that the solution p exists and is unique. Nevertheless, we give here a uniqueness proof that does not involve any specific basis, primarily because it uses an argument we will use frequently.

Theorem 6.4. *The solution of the interpolation problem is unique.*

Proof. For contradiction, assume there exist two distinct polynomials $p, q \in \mathcal{P}_n$ verifying $p(x_i) = y_i$ and $q(x_i) = y_i$ for $i = 0, \ldots, n$. Then, the polynomial $h = p - q$ is also in $\mathcal{P}_n$ and it has $n + 1$ roots:

$$h(x_i) = p(x_i) - q(x_i) = 0 \text{ for } i = 0, \ldots, n.$$

Yet, the only polynomial of degree at most n which has strictly more than n roots is the zero polynomial. Thus, $h = 0$, and it follows that $p = q$. □

If the data points (x_i, y_i) are obtained by sampling a function f, that is, $y_i = f(x_i)$, and $x_0, \ldots, x_n$ are distinct points in $[a, b]$, then, a natural question is:

> *How large can the error $f(x) - p_n(x)$ be for $x \in [a, b]$, where p_n is the interpolation polynomial for $n + 1$ points?*

This is actually a polynomial approximation question: something we will talk about extensively in later lectures. One key observation is that the approximation error depends of course on f itself, but it also very much depends on the choice of points x_i, as we now illustrate with two functions:

$$f_1(x) = \frac{\cos(x) + 1}{2} \text{ over } [0, 2\pi], \text{ and} \tag{6.7}$$

$$f_2(x) = \frac{1}{1 + x^2} \text{ over } [-5, 5]. \tag{6.8}$$

Both are infinitely continuously differentiable. Yet, one will prove much harder to approximate than the other. Run the lecture code to observe the following:

- With equispaced points, at first, increasing n yields better approximation quality for f_1, until a turning point is hit and approximation deteriorates.
- With equispaced points, for f_2, increasing n seems to deteriorate the approximation quality right away: none of the polynomials attain small approximation error. This is the *Runge phenomenon*.
- In both cases, the errors seem largest close to the boundaries of $[a, b]$, which suggests sampling more points there. This is the idea behind Chebyshev points, which we will justify later.
- With Chebyshev points, approximation quality for f_1 improves with n until we reach machine precision, at which point the error stabilizes.
- For f_2, Chebyshev points again allow approximation error to decrease with n, albeit more slowly.

The following theorem bounds the approximation error, allowing to understand the separate roles of f and the interpolation points x_i. We will use this theorem over and over from now on.

Theorem 6.5 (Theorem 6.2 in [SM03])**.** *Let $f\colon [a, b] \to \mathbb{R}$ be $n + 1$ times continuously differentiable. Let $p_n \in \mathcal{P}_n$ be the interpolation polynomial for f at $x_0, \ldots, x_n$ distinct in $[a, b]$. Then, for any $x \in [a, b]$, there exists $\xi \in (a, b)$ (which may depend on x) such that*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\pi_{n+1}(x), \tag{6.9}$$

where $\pi_{n+1}(x) = (x - x_0) \cdots (x - x_n)$. *Defining* $M_{n+1} = \max_{\xi \in [a,b]} |f^{(n+1)}(\xi)|$,

$$|f(x) - p_n(x)| \leq \frac{M_{n+1}}{(n+1)!} |\pi_{n+1}(x)| \tag{6.10}$$

for any $x \in [a, b]$.

Notice that the dependence on f is only through M_{n+1}, while the dependence on x_i's is only through π_{n+1}. In general, f is forced by the application and we have little or no control over it. On the other hand, we often have control over the interpolation points. Thus, it makes sense to try and find x_i's which keep π_{n+1} small in some appropriate sense. Later, we will show that this is precisely what the Chebyshev points aim to do. For now, we just give a bound on π_{n+1} for equispaced points, since it is such a natural case to consider.

Lemma 6.6. *For equispaced points* $x_0, \ldots, x_n$ *on* $[a, b]$ *with* $x_0 = a, x_n = b$, *the spacing between any two points is* $h = \frac{b-a}{n}$ *and*

$$|\pi_{n+1}(x)| \leq \frac{n! h^{n+1}}{4} \tag{6.11}$$

for all $x \in [a, b]$.

Proof. The statement is clear if x is equal to one of the x_i's. Thus, consider $x \neq x_i$ for all i. Hence, x lies strictly between two consecutive points, that is, there exists i such that $x_i < x < x_{i+1}$. Considering the two terms of π_{n+1} pertaining to these two points, since they form a quadratic, we easily obtain:

$$|(x - x_i)(x - x_{i+1})| \leq \max_{\xi \in [x_i, x_{i+1}]} |(\xi - x_i)(\xi - x_{i+1})| = \frac{h^2}{4}.$$

(Indeed, the maximum of the quadratic is attained in the middle of the two roots.) We now need to consider all other terms in π_{n+1}. There are two kinds: for $k < i$, we have $x_k < x_i < x < x_{i+1}$; on the other hand, for $k > i + 1$, we have $x_k > x_{i+1} > x > x_i$. Hence,

$$|x_k - x| \leq \begin{cases} |x_{i+1} - x_k| = (i + 1 - k)h & \text{if } k < i, \\ |x_k - x_i| = (k - i)h & \text{if } k > i + 1. \end{cases}$$

(This is clearer if you simply make a drawing of the situation on the real line.) Combine all inequalities to control π_{n+1}:

$$\begin{aligned} |\pi_{n+1}(x)| &= |x - x_0| \cdots |x - x_{i-1}| |(x - x_i)(x - x_{i+1})| |x - x_{i+2}| \cdots |x - x_n| \\ &\leq (i + 1) \cdots 2 \cdot \frac{1}{4} \cdot 2 \cdots (n - i) h^{n+1}. \end{aligned}$$

The product of integers attains its largest value if x lies in one of the extreme intervals: (x_0, x_1) or (x_{n-1}, x_n), that is, $i = 0$ or $i = n - 1$. Hence,

$$|\pi_{n+1}(x)| \leq \frac{n!}{4} h^{n+1}$$

for all $x \in [a, b]$. □

Combining with Theorem 6.5, a direct corollary is that for equispaced points the approximation error is bounded by $\frac{h^{n+1}}{4(n+1)} M_{n+1}$. As n increases, the fraction decreases exponentially fast. Importantly, the function-dependent M_{n+1} can increase with n, possibly fast enough to still push the bound to infinity. This in itself does not imply the actual error will go to infinity, but it is indicative that large errors are possible.

Question 6.7. *For f_1 (6.7), what is a good value for M_{n+1}? Based on this, what is your best explanation for the experimental behavior of the approximation error?*

■

We now give a proof of the main theorem, following [SM03, Thm 6.2].

Proof of Theorem 6.5. It is only necessary to establish equation (6.9). If x is equal to any of the interpolation points x_i, that equation clearly holds. Thus, it remains to prove (6.9) holds for $x \in [a, b]$ *distinct* from any of the interpolation points. To this end, consider the following function on $[a, b]$:

$$\phi(t) = f(t) - p_n(t) - \frac{f(x) - p_n(x)}{\pi_{n+1}(x)} \pi_{n+1}(t).$$

(Note that x is fixed in this definition: ϕ is only a function of t.) Here is are the two crucial observations:

1. ϕ has at least $n + 2$ distinct roots in $[a, b]$. Indeed, $\phi(x_i) = 0$ for $i = 0, \ldots, n$, and also $\phi(x) = 0$, and

2. ϕ is $n + 1$ times continuously differentiable.

Using both facts, by the mean value theorem (or Rolle's theorem), the derivative of ϕ has at least $n+1$ distinct roots in $[a, b]$. In turn, the second derivative of ϕ has at least n distinct roots in $[a, b]$. Continuing this argument, we conclude that the $(n + 1)$st derivative of ϕ (which we denote by $\phi^{(n+1)}$) has at least one root in $[a, b]$: let us call it ξ. (Of course, ξ may depend on x in a complicated way, but that is not important: we only need to know that such a ξ exists.) Verify the two following claims:

1. The $(n+1)$st derivative of $p_n(t)$ with respect to t is identically 0, and
2. The $(n+1)$st derivative of $\pi_{n+1}(t) = t^{n+1} + (\text{lower order terms})$ with respect to t is $(n+1)!$ (constant).

Then, the $(n+1)$st derivative of ϕ is:

$$\phi^{(n+1)}(t) = f^{(n+1)}(t) - \frac{f(x) - p_n(x)}{\pi_{n+1}(x)}(n+1)!.$$

Since $\phi^{(n+1)}(\xi) = 0$, we conclude that

$$0 = f^{(n+1)}(\xi) - \frac{f(x) - p_n(x)}{\pi_{n+1}(x)}(n+1)!,$$

hence (6.9) holds for all $x \in [a, b]$. □

6.2 Hermite interpolation

Hermite interpolation is a variation of the Lagrange interpolation problem. It is stated as follows.

Problem 6.8. *Given $x_0, \ldots, x_n \in \mathbb{R}$ distinct and $y_0, \ldots, y_n \in \mathbb{R}, z_0, \ldots, z_n \in \mathbb{R}$, find a polynomial p of degree at most $2n+1$ such that $p(x_i) = y_i$ and $p'(x_i) = z_i$ for $i = 0, \ldots, n$.*

This problem is solved following the same construction strategy. With the notation $\delta_{ik} = 1$ if $i = k$ and $\delta_{ik} = 0$ if $i \neq k$, we must:

1. Find $H_0, \ldots, H_n \in \mathcal{P}_{2n+1}$ such that $H_k(x_i) = \delta_{ik}$ and $H_k'(x_i) = 0$ for all i, k; and
2. Find $K_0, \ldots, K_n \in \mathcal{P}_{2n+1}$ such that $K_k(x_i) = 0$ and $K_k'(x_i) = \delta_{ik}$ for all i, k.

Then, by linearity, the solution p to the Hermite problem is:

$$p = \sum_{k=0}^{n} y_k H_k + z_k K_k.$$

To construct the basis polynomials H_k, K_k, it is easiest to work our way up from the Lagrange polynomials,

$$L_k(x) = \prod_{i \neq k} \frac{x - x_i}{x_k - x_i}.$$

For example, K_k must have a double root at each $x_{i\neq k}$ and a simple root at x_k: this accounts for all the $2n+1$ possible roots of K_k, thus it must be that

$$K_k(x) \propto (L_k(x))^2 (x - x_k).$$

The scale is set by enforcing $K_k'(x_k) = 1$. By chance, the above is already properly scaled: it is an equality. To construct H_k, one can work similarly (though it's a tad less easy to guess), and one obtains:

$$H_k(x) = (L_k(x))^2 \left(1 - 2L_k'(x_k)(x - x_k)\right).$$

Here too, it is an exercise to verify that H_k satisfies its defining properties.

A theorem similar to Theorem 6.5 can be established for Hermite interpolation too. We omit the proof.

Theorem 6.9 (Theorem 6.4 in [SM03])**.** *Let $f\colon [a,b] \to \mathbb{R}$ be $2n+2$ times continuously differentiable. Let $p_{2n+1} \in \mathcal{P}_{2n+1}$ be the Hermite interpolation polynomial for f at $x_0, \ldots, x_n$ distinct in $[a,b]$. Then, for any $x \in [a,b]$, there exists $\xi \in (a,b)$ (which may depend on x) such that*

$$f(x) - p_{2n+1}(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}(\pi_{n+1}(x))^2, \tag{6.12}$$

where $\pi_{n+1}(x) = (x-x_0)\cdots(x-x_n)$. Defining $M_{2n+2} = \max_{\xi\in[a,b]} |f^{(2n+2)}(\xi)|$,

$$|f(x) - p_n(x)| \leq \frac{M_{2n+2}}{(2n+2)!}(\pi_{n+1}(x))^2 \tag{6.13}$$

for any $x \in [a,b]$.

Notice that the dependence on the interpolation point is also through π_{n+1}—same as for Lagrange interpolation. Thus, if we find interpolation points which make $|\pi_{n+1}|$ small, these will be relevant for both Lagrange and Hermite interpolation. This theorem can be used to understand errors in Gauss quadrature for numerical integration (a problem we discuss later.)

Chapter 7

Minimax approximation

In the previous chapter, we studied polynomial interpolation. Theorem 6.5 went a bit beyond the original intent, venturing into polynomial approximation. Indeed, it states that for f sufficiently smooth, the approximation error of p_n at *all* points in the interval $[a, b]$ is bounded as

$$|f(x) - p_n(x)| \leq \frac{M_{n+1}}{(n+1)!}|\pi_{n+1}(x)|. \tag{7.1}$$

A more general way to look at this result is to introduce the ∞-norm, and look at this result as saying that some particular norm of $f - p_n$ is bounded. We do this now.

Definition 7.1. *Let $C[a, b]$ denote the set of real-valued continuous functions on $[a, b]$. This is an infinite-dimensional linear space. The ∞-norm of a function $g \in C[a, b]$ is defined as:*

$$\|g\|_\infty = \max_{x \in [a,b]} |g(x)|.$$

This is well defined since $|g|$ is continuous on the compact set $[a, b]$ (Weierstrass). It is an exercise to verify this is indeed a norm.

With this notion of norm, we can express (a slightly relaxed version of)[1] the error bound above as:

$$\|f - p_n\|_\infty \leq \frac{M_{n+1}}{(n+1)!}\|\pi_{n+1}\|_\infty. \tag{7.2}$$

Furthermore, we have the convenient notation $M_{n+1} = \|f^{(n+1)}\|_\infty$.

This formalism leads to three natural questions:

[1] The error bound (7.1) specifies it is the same x on both sides of the inequality, whereas (7.2) takes the worst case on both sides independently.

1. Can we pick $x_0, \ldots, x_n$ such that $\|\pi_{n+1}\|_\infty$ is minimal? Then, interpolating at those points is a good idea, in particular if we do not know much about f. We will solve this question completely and explicitly.

2. If the goal is to approximate f, who says we must interpolate? We could try to solve this directly:

$$\min_{p_n \in \mathcal{P}_n} \|f - p_n\|_\infty.$$

   That is: minimize the actual error rather than a bound on the error. This is the central question of this chapter. We will characterize the solutions (show existence, uniqueness and more), but we won't do much in the way of computations.

3. What about other norms?

Question 2 puts the finger on the central problem of this chapter.

Problem 7.2. *Given $f \in C[a,b]$, find $p_n \in \mathcal{P}_n$ such that*

$$\begin{aligned} \|f - p_n\|_\infty &= \min_{q \in \mathcal{P}_n} \|f - q\|_\infty \\ &= \min_{q \in \mathcal{P}_n} \max_{x \in [a,b]} |f(x) - q(x)|. \end{aligned}$$

The solution to this problem is called the minimax polynomial *for f on $[a,b]$, because of the min-max combination.*

Notice that, at this stage, it is unclear whether minimax polynomials exist and whether they are unique—we will establish this as we go. Furthermore, it may very well be that the minimax p_n interpolates f at some points (indeed, that will be the case), but this is not required a priori.

Answering Question 2 provides an answer to **Question 1**. Indeed, consider this expansion of π_{n+1}:

$$\pi_{n+1}(x) = (x - x_0) \cdots (x - x_n) = x^{n+1} - q_n(x),$$

where $q_n \in \mathcal{P}_n$. Minimizing $\|\pi_{n+1}\|_\infty$ is equivalent to minimizing $\|x^{n+1} - q_n\|_\infty$, which is exactly the problem of finding the minimax approximation of $f(x) = x^{n+1}$ in $\mathcal{P}_n$.[2]

[2]It is unclear at this point that picking q_n as such leads to $x^{n+1} - q_n(x)$ having $n+1$ distinct real roots in $[a,b]$, which is required to factor π_{n+1} into $(x - x_0) \cdots (x - x_n)$—this will be resolved fully.

Last but not least, regarding **Question 3**: we could choose to minimize the error $\|f - p_n\|$ for some other norm. In particular, in the next chapter we consider the *(weighted) 2-norm* over $C[a, b]$:

$$\|g\|_2 = \sqrt{\int_a^b w(x)|g(x)|^2 \mathrm{d}x}, \tag{7.3}$$

where w is a proper weight function (to be discussed in the next chapter.) Importantly, in infinite dimensional spaces such as $C[a, b]$, *the choice of norm matters*. Remember, in *finite* dimensional spaces such as $\mathbb{R}^n$, all norms are equivalent in that for any two norms $\|\cdot\|_\alpha, \|\cdot\|_\beta$ there exist constants $c_1, c_2 > 0$ such that

$$\forall x \in \mathbb{R}^n, \qquad c_1\|x\|_\beta \leq \|x\|_\alpha \leq c_2\|x\|_\beta.$$

This notably means convergence in one norm implies convergence in all norms. Not so in infinite dimensional spaces! This is partly why we have two whole, separate chapters about polynomial approximation in two different norms: they are quite different problems, requiring different mathematics.

Question 7.3. *Propose a sequence of functions $f_1, f_2, \ldots$ in $C[-1, 1]$ such that $\|f_n\|_2 \to 0$ (with $w(x) \equiv 1$) whereas $\|f_n\|_\infty \to \infty$.*

■

Question 7.4. *Show that the opposite cannot happen. Specifically, show there exists a constant c such that $\|f\|_2 \leq c\|f\|_\infty$.*

■

Questions beget questions. A fourth comes to mind:

4. How low can we go? Can we truly expect *any* continuous function to be arbitrarily well approximated by some polynomial?

The answer to this question is: yes! This is the message of the following fundamental result (proof omitted.)

Theorem 7.5 (Weierstrass approximation theorem, Theorem 8.1 in [SM03])**.** *For any $f \in C[a, b]$, for any tolerance $\varepsilon > 0$, there exists a polynomial p such that*

$$\|f - p_n\|_\infty \leq \varepsilon.$$

In other words, polynomials are dense *in $C[a, b]$ (in the same sense that rational numbers are dense among the reals.)*

The catch is: the theorem does not specify how large the degree of p may need to be as a function of f and ε. Admittedly, it could be impractically large in general. In what follows, we maintain some control over the degree, using results with the same flavor as Theorem 6.5. Note that the above result extends to $\|\cdot\|_2$ directly using Question 7.4.

7.1 Characterizing the minimax polynomial

How do we compute the minimax polynomial? This is a tough question. The classical algorithm for this is the *Remez exchange algorithm*.[3] We won't study this algorithm here, in good part because actual minimax polynomials are harder to compute than interpolants at Chebyshev nodes, while the latter are usually good enough. Yet, we will study minimax approximation sufficiently to characterize solutions. This in turn will allow us to construct the Chebyshev nodes and to understand why they are such a good choice.

Let's start with the basics.

Theorem 7.6 (Existence, Theorem 8.2 in [SM03])**.** *Given $f \in C[a,b]$, there exists $p_n \in \mathcal{P}_n$ such that*

$$\|f - p_n\|_\infty \leq \|f - q\|_\infty \qquad \forall q \in \mathcal{P}_n.$$

Before we get to the proof, you may wonder: why does this necessitate a proof at all? It all hinges upon the distinction between *infimum* and *minimum*.[4] The infimum of a set of reals is always well defined: it is the largest lower-bound on the elements of the set. In contrast, the minimum is only defined if the infimum is *equal* to some element in the set; we then say the infimum or minimum is *attained*. In our scenario, each polynomial $q \in \mathcal{P}_n$ maps to a real number $\|f - q\|_\infty$. The set of those numbers necessarily has an infimum. The question is whether some number in that set is equal to the infimum, that is, whether there exists a polynomial p_n in $\mathcal{P}_n$ such that $\|f - p_n\|_\infty$ is *equal* to the infimum. If that is the case, the infimum is called the minimum, and p_n is called a minimizer. This existence theorem is all about that particular point.

Proof. Minimax approximation is an optimization problem:

$$\min_{q \in \mathcal{P}_n} E(q),$$

[3] https://en.wikipedia.org/wiki/Remez_algorithm

[4] https://math.stackexchange.com/questions/342749/what-is-the-difference-between-minimum-and-infimum

where $E\colon \mathcal{P}_n \to \mathbb{R}$ is the cost function: $E(q) = \|f-q\|_\infty$. Our task is to show that the min is indeed a min and not an inf, that is: we need to show the minimum value of E exists and is attained for some q. The usual theorem one aims to invoke for such matters is Weierstrass' Extreme Value Theorem:[5]

> *Continuous functions on non-empty compact sets attain their bounds. (EVT)*

Thus, we contemplate two questions:

1. Is E continuous?
2. $\mathcal{P}_n$ is finite dimensional. In finite dimensional subspaces, a set is compact if and only if it is closed and bounded.[6] $\mathcal{P}_n$ is not bounded, hence it is not compact; can we resolve that?

Continuity of E is clear: continuous perturbations of q lead to continuous perturbations of $|f-q|$, which in turn lead to continuous perturbations of the maximum value of $|f-q|$ on $[a,b]$, which is $E(q)$ (details omitted).

The key point is to address non-compactness of $\mathcal{P}_n$. To this end, the strategy is to construct a compact set $S \subset \mathcal{P}_n$ such that

$$\inf_{q\in\mathcal{P}_n} E(q) = \inf_{q\in S} E(q) \underbrace{=}_{\text{Weierstrass EVT}} \min_{q\in S} E(q). \tag{7.4}$$

If we prove the first equality, then the above states the infimum over $\mathcal{P}_n$ is equivalent to a min over S, thus showing existence of an optimal q.

Define S as follows:

$$S = \{q \in \mathcal{P}_n : E(q) \le \|f\|_\infty\}.$$

This set is indeed bounded and closed, hence it is compact. Furthermore, $0 \in S$ (that is, the zero polynomial is in S) since $0 \in \mathcal{P}_n$ and

$$E(0) = \|f\|_\infty \le \|f\|_\infty.$$

Thus, S is non-empty: EVT indeed applies to S.

It remains to show $\inf_{q\in\mathcal{P}_n} E(q) = \inf_{q\in S} E(q)$. To this end, let $q^* \in S$ be a minimizer for E in S. Then, since $0 \in S$,

$$E(q^*) = \min_{q\in S} E(q) \le E(0) = \|f\|_\infty.$$

[5] https://en.wikipedia.org/wiki/Extreme_value_theorem#Generalization_to_metric_and_topological_spaces

[6] That's another Weierstrass theorem: Bolzano–Weierstrass, https://en.wikipedia.org/wiki/Bolzano%E2%80%93Weierstrass_theorem.

As a result, for any $q \notin S$ we have $E(q) > \|f\|_\infty \geq E(q^*)$. Since $\inf_{q\in\mathcal{P}_n} E(q)$ is the largest lower bound on all values attained by E in $\mathcal{P}_n$, and since all values attained by E outside of S are strictly larger than the smallest value attained in S, it follows that (7.4) holds indeed. □

Given a polynomial $r \in \mathcal{P}_n$, it is a priori quite difficult to determine if this polynomial is minimax, let alone to quantify *how far* it is from being minimax. The following theorem resolves most[7] of the difficulty.

Theorem 7.7 (De la Vallée Poussin's theorem, Theorem 8.3 in [SM03])**.** *Let $f \in C[a,b]$ and $r \in \mathcal{P}_n$. Suppose there exist $n+2$ points $x_0 < \cdots < x_{n+1}$ in $[a,b]$ such that*

$$f(x_i) - r(x_i) \qquad \text{and} \qquad f(x_{i+1}) - r(x_{i+1})$$

have opposite signs for $i = 0, \ldots, n$.[8] *Then,*

$$\min_{i=0,\ldots,n+1} |f(x_i) - r(x_i)| \leq \min_{q\in\mathcal{P}_n} \|f - q\|_\infty \leq \|f - r\|_\infty. \tag{7.5}$$

The upper bound $\min_{q\in\mathcal{P}_n} \|f - q\|_\infty \leq \|f - r\|_\infty$, is trivial. It is the lower bound which is informative and non trivial. In particular, if the lower bound and the upper bound approximately match, this tells us r is *almost* minimax. This is powerful information. Of course, there is no reason a priori to believe we can make the gap small, but we will see later that a polynomial is minimax if and only if the lower and upper bounds match (provided the best possible x_i's are picked for the lower bound.) See also Figure 7.1.

Proof. Define μ as the left-hand side of our inequality of interest:

$$\mu = \min_{i=0,\ldots,n+1} |f(x_i) - r(x_i)|.$$

Certainly, if $\mu = 0$, then the theorem is true. Hence, we only need to consider the case $\mu > 0$. Let $p_n \in \mathcal{P}_n$ be minimax for f on $[a,b]$ (we proved its existence above). For contradiction, let us assume that

$$\|f - p_n\|_\infty < \mu.$$

Our goal is to derive a contradiction from that statement. To this end, consider the following inequalities, valid for all $i = 0, \ldots, n+1$:

$$|p_n(x_i) - f(x_i)| \leq \|f - p_n\|_\infty < \mu \leq |r(x_i) - f(x_i)|. \tag{7.6}$$

[7] Most but not all, because the condition on r is not satisfied in general.

[8] If $f(x_i) - r(x_i) = 0$ for some i, then the theorem is trivially true regardless of the meaning given to the words "opposite signs" in the presence of 0's, so that there is no need to be more specific.

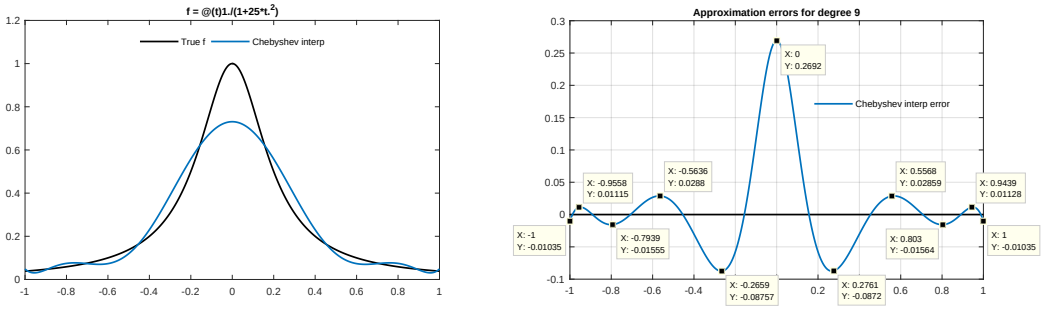


Figure 7.1: Chebyshev interpolation at $n+1 = 10$ points of the Runge function $f(x) = \frac{1}{1+25x^2}$ for $x \in [-1, 1]$. Consider the oscillations of the error in light of De la Vallée Poussin's theorem. Compare with Figure 7.2.

Now consider the difference $r - p_n$ at these special points:

$$r(x_i) - p_n(x_i) = (r(x_i) - f(x_i)) - (p_n(x_i) - f(x_i)) .$$

In inequality (7.6), we showed that, in absolute value, the first term is strictly larger than the second term. Thus,

$$\operatorname{sign}(r(x_i) - p_n(x_i)) = \operatorname{sign}(r(x_i) - f(x_i)) .$$

By assumption, the right-hand side changes sign $n+1$ times as $i = 0, \ldots, n$. But the left-hand side involves $r - p_n$: a polynomial of degree n. If that polynomial changes sign $n+1$ times, the intermediate value theorem implies it has at least $n+1$ distinct roots; but since it has degree at most n, this is only possible if $r(x) = p_n(x)$ for all x. This is a contradiction since $r(x_i) - p_n(x_i) \neq 0$ for all i. Alternatively, we can also notice that if $r = p_n$, then $\|f - r\|_\infty = \|f - p_n\|_\infty < \mu$, yet by (7.5) we know $\|f - r\|_\infty \geq \mu$: this is also a contradiction. □

Having De la Vallée Poussin's theorem at our disposal, one direction of the following equivalence theorem becomes easy to prove. An equivalence theorem is a powerful thing: it tells us that two seemingly different sets of properties (being minimax on the one hand, admitting a special sequence $\{x_0, \ldots, x_{n+1}\}$ on the other hand) are actually equivalent. Pragmatically, this means every time we are searching for or dealing with a minimax theorem, we can invoke those sequences without loss. The particular form of this theorem also makes it a lot easier to verify whether any given polynomial is or is not minimax—a non trivial feat.

Theorem 7.8 (Oscillation theorem, Theorem 8.4 in [SM03])**.** *Consider $f \in C[a,b]$. A polynomial $r \in \mathcal{P}_n$ is minimax for f on $[a,b]$* if and only if *there exist $n+2$ points $x_0 < \cdots < x_{n+1}$ in $[a,b]$ such that*

1. $|f(x_i) - r(x_i)| = \|f - r\|_\infty$ *for* $i = 0, \ldots, n+1$, *and*
2. $f(x_i) - r(x_i) = -(f(x_{i+1}) - r(x_{i+1}))$ *for* $i = 0, \ldots, n$.

Proof. The proof is in two parts:

- Sufficiency: We assume $x_0 < \cdots < x_{n+1}$ are given and satisfy the conditions of the theorem; we aim to show r is minimax. Apply De la Vallée Poussin's theorem:

$$\|f - r\|_\infty = \min_{i=0,\ldots,n+1} |f(x_i) - r(x_i)| \overset{\text{DlVP}}{\leq} \min_{q \in \mathcal{P}_n} \|f - q\|_\infty \leq \|f - r\|_\infty.$$

  The left-most and right-most quantities are the same. We conclude inequalities must be equalities:

$$\|f - r\|_\infty = \min_{q \in \mathcal{P}_n} \|f - q\|_\infty,$$

  that is, r is minimax.

- Necessity: This is the technical part of the proof. We omit it—see the reference book for a full account.[9] □

The oscillation theorem leads to uniqueness of the minimal polynomial (using the necessity part of the theorem.)

Theorem 7.9 (Uniqueness, Theorem 8.5 in [SM03])**.** *Each $f \in C[a,b]$ admits a unique minimax polynomial in $\mathcal{P}_n$.*

Proof. The proof is by contradiction. Assume $p_n, q_n \in \mathcal{P}_n$ are two *distinct* minimax polynomials for f. Then, we argue

1. $\frac{p_n + q_n}{2}$ is also minimax, and
2. this implies p_n, q_n coincide at $n+2$ distinct points, showing $p_n = q_n$.

Here are the arguments for both parts:

[9] A PDF version of [SM03] has some errors that were corrected in the print version, specifically for the construction of x_0.

1. Let $E = \|f - p_n\|_\infty = \|f - q_n\|_\infty$. Then,

$$E \leq \left\| f - \frac{p_n + q_n}{2} \right\|_\infty = \frac{1}{2}\|f - p_n + f - q_n\|_\infty \\ \leq \frac{1}{2}(\|f - p_n\|_\infty + \|f - q_n\|_\infty) = E.$$

2. Since $\frac{p_n+q_n}{2} \in \mathcal{P}_n$ is minimax, the oscillation theorem gives $x_0 < \cdots < x_{n+1}$ in $[a, b]$ such that (first property)

$$\left| f(x_i) - \frac{p_n(x_i) + q_n(x_i)}{2} \right| = E \quad \forall i.$$

Multiply by 2 on each side:

$$|\underbrace{f(x_i) - p_n(x_i)}_{|\cdot| \leq E} + \underbrace{f(x_i) - q_n(x_i)}_{|\cdot| \leq E}| = 2E \quad \forall i.$$

This implies $f(x_i) - p_n(x_i) = f(x_i) - q_n(x_i)$ for each i, in turn showing $p_n(x_i) = q_n(x_i)$ for each i. Thus, the polynomial $p_n - q_n \in \mathcal{P}_n$ has $n+2$ distinct roots: this can only happen if $p_n = q_n$. □

7.2 Interpolation points to minimize the bound

We can leverage all of this new insight to determine the best interpolation points to minimize the upper bound on the approximation error we determined in Theorem 6.5. Indeed, recall that for $f \in C[a, b]$ continuously differentiable $n + 1$ times we established

$$\|f - p_n\|_\infty \leq \frac{M_{n+1}}{(n+1)!}\|\pi_{n+1}\|_\infty,$$

where $p_n \in \mathcal{P}_n$ interpolates f at $x_0 < \cdots < x_n$ in $[a, b]$, $M_{n+1} = \|f^{(n+1)}\|_\infty$ and

$$\pi_{n+1}(x) = (x - x_0) \cdots (x - x_n) = x^{n+1} - q_n(x).$$

Above, $q_n \in \mathcal{P}_n$ depends on the choice of interpolation points. Our goal is to pick $x_0, \ldots, x_n$ so as to minimize

$$\|\pi_{n+1}\|_\infty = \|x^{n+1} - q_n\|_\infty.$$

The polynomial q_n which minimizes that quantity is the minimax approximation of x^{n+1} over $[a, b]$ in $\mathcal{P}_n$.

Choosing q_n to be this minimax polynomial is a valid strategy only if $x^{n+1} - q_n$ indeed has $n + 1$ distinct real roots in $[a, b]$. Fortunately, the oscillation theorem guarantees this (and more). Indeed, by the oscillation theorem, there exists a sequence of $n + 2$ points in $[a, b]$ for which $x^{n+1} - q_n(x)$ attains the value $\pm\|x^{n+1} - q_n\|_\infty$ with alternating signs. In particular, $x^{n+1} - q_n(x)$ changes sign $n+1$ times in $[a, b]$, which means $x^{n+1} - q_n(x)$ has at least $n + 1$ distinct roots in $[a, b]$. Since $x^{n+1} - q_n(x)$ has degree $n + 1$, these are all of the roots.

We claim that solving the following task is sufficient to find q_n (and the interpolation points) explicitly. Without loss of generality, fix $[a, b] = [-1, 1]$.[10]

Task. Find a sequence of polynomials $T_0, T_1, T_2, \ldots$ such that

1. $T_n \in \mathcal{P}_n \backslash \mathcal{P}_{n-1}$ (degree exactly n),
2. $\|T_n\|_\infty = 1$,
3. $T_n(x)$ attains ± 1 at $n + 1$ points in $[-1, 1]$, alternating.

Why would that work? Write

$$T_n(x) = a_n x^n + \text{ lower order terms}$$

so that a_n denotes the coefficient of the leading order term in T_n (and a_{n-1} denotes the coefficient of the leading order term in T_{n-1}, etc.) Then, define

$$q_n(x) = x^{n+1} - \frac{1}{a_{n+1}} T_{n+1}(x).$$

Crucially, this q_n indeed has degree n, even though it was built from two polynomials of degree $n + 1$. We argue q_n is minimax for x^{n+1}, as desired. Indeed, consider the error function:

$$x^{n+1} - q_n(x) = \frac{1}{a_{n+1}} T_{n+1}(x).$$

[10] To work in a different interval $[a, b]$, simply execute the affine change of variable $t \mapsto \frac{a+b}{2} + \frac{b-a}{2} t$, so that -1 is mapped to a and $+1$ is mapped to b.

By definition of T_{n+1}, the error alternates $n+2$ times between $\pm\frac{1}{a_{n+1}}$, which coincides with $\pm\|x^{n+1}-q_n\|_\infty$. By the oscillation theorem (the part with the easy proof), this guarantees we found a minimax. In conclusion, with

$$(x-x_0)\cdots(x-x_n)=\pi_{n+1}(x)=x^{n+1}-q_n(x)=\frac{1}{a_{n+1}}T_{n+1}(x), \qquad (7.7)$$

we find that picking the $n+1$ interpolation points as the roots of T_{n+1} yields a bound on the approximation error as

$$\|f-p_n\|_\infty \le \frac{M_{n+1}}{(n+1)!}\frac{1}{|a_{n+1}|},$$

since $\|\pi_{n+1}\|_\infty = \|T_{n+1}/a_{n+1}\|_\infty = 1/|a_{n+1}|$. It remains to construct the polynomial T_{n+1} and to determine its roots and a_{n+1}.

Building the T_n's. The T_n's are remarkable polynomials.[11] Let's give them a name.

Definition 7.10. *The* Chebyshev polynomials *are defined by recurrence as:*

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{n+1}(x) &= 2xT_n(x)-T_{n-1}(x) \qquad \textit{for } n=1,2,\ldots \end{aligned}$$

(These are also called the Chebyshev polynomials of the first kind.*)*

We need to

1. Check that these polynomials indeed fulfill our set task; and
2. Investigate a_{n+1} and the roots of T_{n+1}.

Parts of this is straightforward from the recurrence relation. For instance, it is clear that each T_n is indeed in $\mathcal{P}_n$. Furthermore, it is clear that

$$\begin{aligned} a_0 &= 1, \\ a_1 &= 1, \\ a_{n+1} &= 2a_n \qquad \text{for } n=1,2,\ldots. \end{aligned}$$

[11] See https://en.wikipedia.org/wiki/Chebyshev_polynomials for the tip of the iceberg.

Explicitly,

$$\begin{aligned} a_0 &= 1, \\ a_{n+1} &= 2^n \qquad \text{for } n = 0, 1, \ldots \end{aligned} \tag{7.8}$$

This notably confirms each T_n is of degree exactly n. Yet, the other statements regarding $\|T_n\|_\infty$, alternation between ± 1 and roots are not straightforward from the recurrence. For this, we need a surprising lemma.

Lemma 7.11. *For all $n \geq 0$, for all $x \in [-1, 1]$,*

$$T_n(x) = \cos(n \arccos(x)).$$

Proof. The proof is by induction. Start by verifying that the statement holds for T_0 and T_1. Then, as induction hypothesis, assume

$$T_k(x) = \cos(k \arccos(x))$$

for $k = 0, \ldots, n$, for all $x \in [-1, 1]$. We aim to show the same holds for T_{n+1}. To this end, recall the trigonometric identity:

$$2 \cos u \cos v = \cos(u + v) + \cos(u - v).$$

Set $u = n\theta$ and $v = \theta$:

$$2 \cos(n\theta) \cos(\theta) = \cos((n + 1)\theta) + \cos((n - 1)\theta).$$

Define $x = \cos(\theta) \in [-1, 1]$. Then, $\theta = \arccos(x)$ and:

$$2x \underbrace{\cos(n \arccos(x))}_{T_n(x)} = \cos((n + 1) \arccos(x)) + \underbrace{\cos((n - 1) \arccos(x))}_{T_{n-1}(x)},$$

where we used the induction hypothesis in the under-braces. Rearranging, we find by the recurrence relation that, for all $x \in [-1, 1]$,

$$\cos((n + 1) \arccos(x)) = 2xT_n(x) - T_{n-1}(x) = T_{n+1}(x),$$

as desired. □

This lemma makes it straightforward to continue our work. In particular,

1. It is clear that $\|T_n\|_\infty = \max_{x \in [-1,1]} |\cos(n \arccos(x))| = 1$; and
2. Alternation is verified since $T_n(x) = \pm 1$ if and only if $n \arccos(x) = k\pi$ for some integer k, which shows alternation at the points $\cos\left(\frac{k\pi}{n}\right)$: for $k = 0, \ldots, n$, these are $n + 1$ distinct points in $[-1, 1]$.

3. The roots are easily determined: $T_{n+1}(x) = 0$ if and only if

$$\cos((n+1)\arccos(x)) = 0$$

with $x \in [-1, 1]$, that is, for

$$x_k = \cos\left(\frac{\frac{\pi}{2} + k\pi}{n+1}\right), \quad k = 0, \ldots, n. \tag{7.9}$$

The above roots are the so-called *Chebyshev nodes* (of the first kind). Plot them to confirm they are more concentrated near the edges of $[-1, 1]$.

Consequences for interpolation. The machinery developed above leads to the following concrete message regarding polynomial approximation of sufficiently smooth functions: working with the interval $[-1, 1]$, if p_n is the (unique) polynomial in $\mathcal{P}_n$ which interpolates f at the Chebyshev nodes (7.9),

$$\|f - p_n\|_\infty \leq \frac{M_{n+1}}{2^n \cdot (n+1)!}.$$

The denominator grows *fast*. It takes a particularly wild smooth function for this bound not to go to zero. Of more direct concern though, note that this bound is only useful if f is indeed sufficiently differentiable. Simple functions such as $f(x) = |x|$ cause trouble of their own. You should check it experimentally. For $f(x) = \text{sign}(x)$ (which is not even continuous, hence breaks the most fundamental of our assumptions), you will witness the Gibbs phenomenon, familiar to those of you who know Fourier transforms.

Question 7.12. *What happens to this bound if we use the affine change of variable to work on* $[a, b]$ *instead of* $[-1, 1]$*?*

■

We close this investigation here (for now at least.) To state the obvious: there is a lot more to Chebyshev nodes and Chebyshev polynomials than meets the eye thus far. Having a look at the Wikipedia page linked above can give you some hints of the richness of this topic. We will encounter some of it in the next chapter, as an instance of *orthogonal polynomials*.

7.3 Codes and figures

The following code uses the Chebfun toolbox.[12] Among many other things relevant to this course, Chebfun notably packs ready-made functions for min-

[12] http://www.chebfun.org

imax approximation (which is the main reason for using it here) and for Lagrange interpolation at Chebyshev nodes (which is used here too, but for which we could have just as easily used our own codes based on the previous chapter.)

```
%%
clear;
close all;
clc;
set(0, 'DefaultLineLineWidth', 2);
set(0, 'DefaultLineMarkerSize', 30);

%% Pick a polynomial degree: play with this parameter
n = 16;

%% Pick a function to play with and an interval [a, b]

% Notice that many functions are even or odd:
% Check carefully effect of picking n even or odd as well.

f = @(t) 1./(1+25*t.^2);
% f = @(t) sin(t).*(t.^2-2);
% f = @(t) sqrt(cos((t+1)/2));
% f = @(t) log(cos(t)+2);
% f = @(t) t.^(n+1);
% f = @abs;
% f = @sign; % Not continuous! Our theorems break down.

a = -1;
b = 1;

% fc is a representation of f in Chebfun format.
% For all intents and purposes, this is the same
% as f up to machine precision.)
%
% You can get the Chebfun toolbox freely at
% http://www.chebfun.org.
fc = chebfun(f, [a, b], 'splitting', 'on');

subplot(1, 2, 1);
plot(fc, 'k-');
subplot(1, 2, 2);
plot([a, b], [0, 0], 'k-');

% Chebyshev interpolation
% Get a polynomial of degree n to approximate f on [a, b]
% through interpolation at the Chebyshev points
```

```
% (of the first kind.)

% The interpolant is here obtained through Chebfun. Manually,
% you can also use the explicit expression for the Chebyshev
% nodes together with your codes regarding Lagrange
% interpolation (barycentric formula) to evaluate the
% interpolant: it's the same.
%
% n+1 indicates we want n+1 nodes (polynomial has degree n).
% chebkind = 1 selects Chebyshev nodes of the first kind.
fn = chebfun(f, [a, b], n+1, 'chebkind', 1);

% plot the polynomial
subplot(1, 2, 1); hold all;
plot(fn);
title(['f = ' char(f)]);

% Plot the approximation error
subplot(1, 2, 2); hold all;
plot(fc-fn);

grid on;
title(sprintf('Approximation errors for degree %d', n));

%% Actual minimax (via Remez algorithm)

pn = minimax(fc, n);

% Plot minimax polynomial
subplot(1, 2, 1); hold all;
plot(pn);

% Plot minimax approximation error
subplot(1, 2, 2); hold all;
plot(fc-pn);


%% Legend
subplot(1, 2, 1);
legend('True f', 'Chebyshev interp', 'Minimax', ...
       'Orientation', 'horizontal', 'Location', 'North');
legend('boxoff');
pbaspect([1.6, 1, 1]);
subplot(1, 2, 2);
legend('0', 'Chebyshev interp', 'Minimax', ...
       'Orientation', 'horizontal', 'Location', 'North');
legend('boxoff');
pbaspect([1.6, 1, 1]);
```

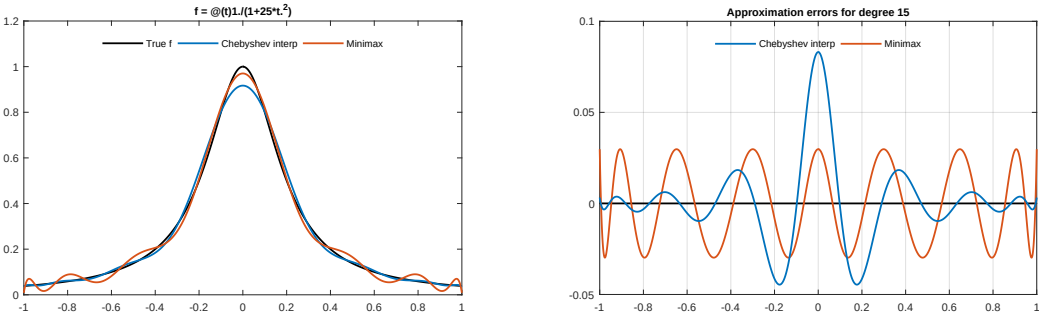


Figure 7.2: Chebyshev interpolation at $n+1 = 16$ points compared to minimax approximation of degree $n = 15$ of the Runge function $f(x) = \frac{1}{1+25x^2}$ for $x \in [-1, 1]$, computed with Chebfun (Remez algorithm.) The Chebyshev approximation is pretty good compared to the optimal minimax, yet is much simpler to compute. If you only had the Chebyshev polynomial, what could you deduce about the minimax from De la Vallée Poussin's theorem?

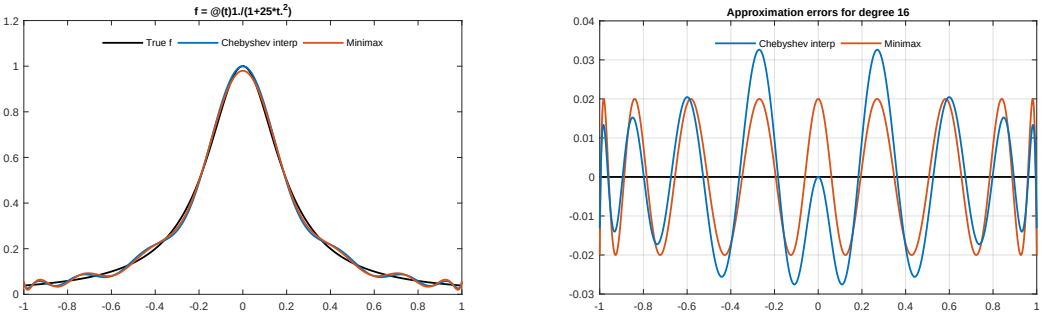


Figure 7.3: Same as Figure 7.2 but with $n = 16$. Since f is even, the minimax polynomial has 19 alternation points rather than only 18 (think about it). If you only had the Chebyshev polynomial, what could you deduce about the minimax from De la Vallée Poussin's theorem?

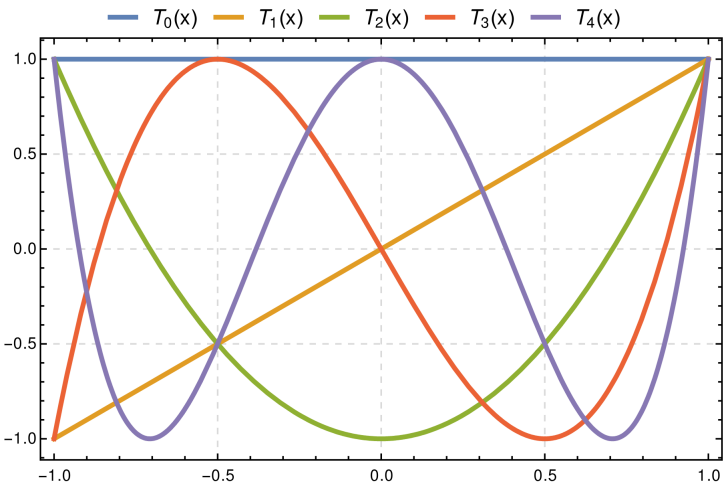


Figure 7.4: The first few Chebyshev polynomials. Picture credit: https://en.wikipedia.org/wiki/Chebyshev_polynomials#/media/File:Chebyshev_Polynomials_of_the_First_Kind.svg.

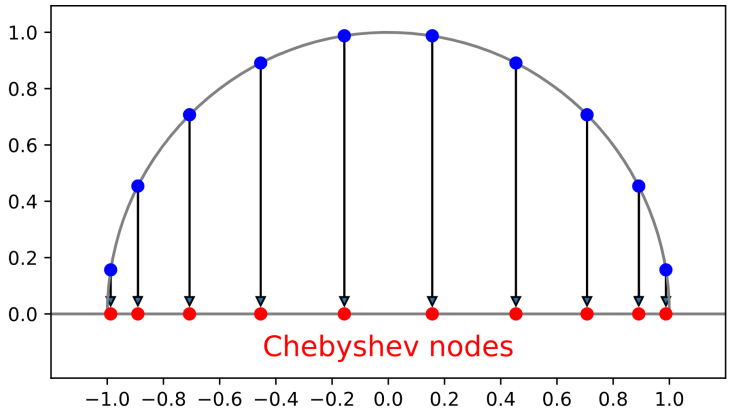


Figure 7.5: Chebyshev nodes are more concentrated toward the ends of the interval, though they come from evenly spaced points on the upper-half of a circle. Picture credit: https://en.wikipedia.org/wiki/Chebyshev_nodes#/media/File:Chebyshev-nodes-by-projection.svg.

Chapter 8

Approximation in the 2-norm

Given a function f on $[a, b]$ (conditions to be specified later), we consider the problem of computing p_n, a solution to

$$\min_{p \in \mathcal{P}_n} \|f - p\|_2,$$

where $\|\cdot\|_2$ is a 2-norm defined below. One key difference with the minimax approximation problem from the previous chapter is that the 2-norm is induced by an inner product (this is not so for the ∞-norm): this will make our life a lot easier.

To begin this chapter, we:

1. Discuss inner products and 2-norms over spaces of functions;
2. Derive the solution to the 2-norm approximation problem, securing existence and uniqueness as a byproduct;
3. Argue that the optimal polynomial p_n really is just the orthogonal projection of f to $\mathcal{P}_n$; and
4. Delve into systems of orthogonal polynomials, which ease such projections.

For the first three elements at least, the math looks very similar to things you have already learned (including in this course) about least squares problems. The crucial point is: we are now working in an *infinite* dimensional space. It is important to go through the steps carefully from first principles, and to keep our intuitions in check.

Remark 8.1. *Take a moment to reflect on this: now and for the last couple of chapters, functions are often considered as vectors. As strange as this may*

sound, remember this: by definition, a vector is nothing but an element of a vector space. The most common vector space being $\mathbb{R}^n$, it is only natural that we tend to forget the more general definition, and conflate in our mind the notion of vector with that of a "column of numbers". It is worth it to take a step back and consider what it means that $C[a,b]$ (for example) is a vector space, and what it means for a function $f \in C[a,b]$ to be a vector in that space. Also think about how this abstraction allows you to use familiar diagrams such as planes and arrows to represent subspaces of $C[a,b]$ ($\mathcal{P}_n$ for example) and functions themselves.

8.1 Inner products and 2-norms

Definition 8.2 (Def. 9.1 in [SM03])**.** *Let V be a linear space over $\mathbb{R}$. A real-valued function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ is an* inner product *on V if*

1. $\langle f, g \rangle = \langle g, f \rangle \quad \forall f, g \in V$;
2. $\langle f + g, h \rangle = \langle f, h \rangle + \langle g, h \rangle \quad \forall f, g, h \in V$;
3. $\langle \lambda f, g \rangle = \lambda \langle f, g \rangle \quad \forall \lambda \in \mathbb{R}, \forall f, g \in V$; *and*
4. $\langle f, f \rangle > 0 \quad \forall f \in V, f \neq 0$.

A linear space with an inner product is called an inner product space.

Definition 8.3 (Def. 9.2 in [SM03])**.** *For $f, g \in V$, if $\langle f, g \rangle = 0$ we say f and g are* orthogonal *(to one another).*

Example 8.4. *If $V = \mathbb{R}^n$ (finite dimensional), the most common inner product by far is:*

$$\langle x, y \rangle = \sum_{k=1}^{n} x_k y_k = x^T y.$$

Generally, for any symmetric and (strictly) positive definite $W \in \mathbb{R}^{n \times n}$,

$$\langle x, y \rangle = x^T W y$$

is an inner product (check the properties above.)

Example 8.5. *For our purpose, the more interesting case is $V = C[a,b]$, the space of continuous functions on $[a,b]$. On that space, given a positive,*

continuous and integrable weight function $w\colon (a,b) \to \mathbb{R}$, *the following is an inner product (check the properties):*

$$\langle f, g\rangle = \int_a^b w(x)f(x)g(x)\mathrm{d}x. \tag{8.1}$$

Notice that w is allowed to take on infinite values at a, b, provided it remains integrable. This will be important later.

Definition 8.6 (Def. 9.3 in [SM03])**.** *An inner product leads to an induced norm (usually called the 2-norm):*

$$\|f\| = \sqrt{\langle f, f\rangle}.$$

Note: it is not trivial that this is indeed a norm (remember, we are in infinite dimensions now; things you learned in finite dimension may not apply.) See [SM03, Lemma 9.1 and Thm. 9.1] for a complete argument based on Cauchy–Schwarz (in infinite dimensions).

Continuing the previous example, we have the following 2-norm over $C[a,b]$:

$$\|f\|_2 = \sqrt{\int_a^b w(x)(f(x))^2\mathrm{d}x}. \tag{8.2}$$

From these expressions for the inner product and the norm, it becomes clear that we do not actually need to restrict ourselves to continuous functions f, as was the case for the ∞-norm. We only need to make sure all integrals that occur are well defined. This is the case over the following linear space:

$$L^2_w(a,b) = \left\{f\colon [a,b] \to \mathbb{R} : w(x)(f(x))^2 \text{ is integrable over } (a,b)\right\}. \tag{8.3}$$

Question 8.7. *Verify $L^2_w(a,b)$ is indeed a linear space, and show that it contains $C[a,b]$ strictly (that is, it contains strictly more than $C[a,b]$.)*

We are finally in a good position to frame the central problem of this chapter.

Problem 8.8. *For a given weight function w on (a,b) and a given function $f \in L^2_w(a,b)$, find $p_n \in \mathcal{P}_n$ such that $\|f - p_n\|_2 \leq \|f - q\|_2$ for all $q \in \mathcal{P}_n$.*

We show momentarily that the solution to this problem exists and is unique. It is called the *polynomial of best approximation* of degree n to f in the 2-norm on (a,b). Since polynomials are dense in $C[a,b]$ for the 2-norm as we discussed in the previous chapter, it is at least the case that for continuous f, as $n \to \infty$, the approximation becomes arbitrarily good. We won't discuss rates of approximation here (that is, how fast the approximation error goes to zero as n increases.)

8.2 Solving the approximation problem

Our unknown is a polynomial of degree at most n. As usual, the first step is to pick a basis:

$$\text{Let } q_0, \ldots, q_n \text{ form a basis for } \mathcal{P}_n.$$

Equipped with this basis, the unknowns are simply the coefficients $c_0, \ldots, c_n$ forming the vector $c \in \mathbb{R}^{n+1}$ such that

$$p_n = c_0 q_0 + \cdots + c_n q_n = \sum_{k=0}^{n} c_k q_k. \tag{8.4}$$

We wish to pick c so as to minimize $\|f - p_n\|_2$. Equivalently, we want to minimize:

$$\begin{aligned} h(c) = \|f - p_n\|_2^2 &= \langle f - p_n, f - p_n \rangle \\ &= \langle f, f \rangle + \langle p_n, p_n \rangle - 2 \langle f, p_n \rangle \\ &= \|f\|_2^2 + \left\langle \sum_k c_k q_k, \sum_\ell c_\ell q_\ell \right\rangle - 2 \left\langle f, \sum_k c_k q_k \right\rangle \\ &= \|f\|_2^2 + \sum_k \sum_\ell c_k c_\ell \langle q_k, q_\ell \rangle - 2 \sum_k c_k \langle f, q_k \rangle . \end{aligned}$$

Introduce the matrix $M \in \mathbb{R}^{(n+1)\times(n+1)}$ and the vector $b \in \mathbb{R}^{n+1}$ defined by:

$$M_{k\ell} = \langle q_k, q_\ell \rangle , \qquad b_k = \langle f, q_k \rangle .$$

Then,

$$\begin{aligned} h(c) = \|f - p_n\|_2^2 &= \|f\|_2^2 + \sum_k \sum_\ell c_k c_\ell M_{k\ell} - 2 \sum_k c_k b_k \\ &= \|f\|_2^2 + c^T M c - 2 b^T c. \end{aligned}$$

This is a quadratic expression in c. How do we minimize it? Let's consider a simpler problem first: how to minimize a quadratic in a single variable? Let

$$h(x) = ax^2 + bx + c.$$

The recipe is well known: if $h''(x) = 2a > 0$ (the function is convex rather than concave, to make sure there indeed is a minimizer), then the unique minimizer of h is such that $h'(x) = 2ax + b = 0$, that is, $x = -\frac{b}{2a}$. This recipe generalizes.

Lemma 8.9. *Let $h\colon \mathbb{R}^n \to \mathbb{R}$ be a smooth function. If $\nabla^2 h(x) \succ 0$ for all x, then the unique minimizer of h is the solution of $\nabla h(x) = 0$, where*

$$(\nabla h(x))_k = \frac{\partial h}{\partial x_k}(x), \qquad (\nabla^2 h(x))_{k,\ell} = \frac{\partial^2 h}{\partial x_k \partial x_\ell}(x).$$

Proof. See chapter about optimization. □

In our case, $h\colon \mathbb{R}^{n+1} \to \mathbb{R}$ is defined by

$$h(c) = \|f - p_n\|_2^2 = \sum_k \sum_\ell M_{k\ell} c_k c_\ell - 2\sum_k b_k c_k + \|f\|_2^2.$$

We need to get the gradient and Hessian. Formally, using the simple rule

$$\frac{\partial c_i}{\partial c_j} = \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases}$$

we get:

$$\begin{aligned}
(\nabla h(c))_j = \frac{\partial h}{\partial c_j}(c) &= \sum_k \sum_\ell M_{k\ell} \frac{\partial}{\partial c_j}(c_k c_\ell) - 2\sum_k b_k \frac{\partial c_k}{\partial c_j} + 0 \\
&= \sum_k \sum_\ell M_{k\ell} \left(\delta_{kj} c_\ell + c_k \delta_{\ell j}\right) - 2\sum_k b_k \delta_{kj} \\
&= \sum_\ell M_{j\ell} c_\ell + \sum_k M_{kj} c_k - 2b_j \\
&= (Mc)_j + (M^T c)_j - 2b_j \\
&= (2Mc - 2b)_j.
\end{aligned}$$

In the last equality, we used that $M = M^T$ since $M_{k\ell} = \langle q_k, q_\ell \rangle$ and inner products are symmetric. Compactly:

$$\nabla h(c) = 2(Mc - b).$$

The Hessian is similarly straightforward to obtain:

$$\begin{aligned}
(\nabla^2 h(c))_{ij} &= \frac{\partial}{\partial c_i} \frac{\partial h}{\partial c_j}(c) \\
&= \frac{\partial}{\partial c_i} \left(\sum_\ell M_{j\ell} c_\ell + \sum_k M_{kj} c_k - 2b_j \right) \\
&= M_{ji} + M_{ij}.
\end{aligned}$$

Hence,

$$\nabla^2 h(c) = 2M.$$

Lemma 8.9 indeed applies in our case since $M \succ 0$.

Question 8.10. *Show that* $M \succ 0$.

■

Consequently, we find that solutions to our problem satisfy $\nabla h(c) = 0$, that is,

$$Mc = b. \tag{8.5}$$

Since M is positive definite, it is in particular invertible. This confirms *existence and uniqueness* of the solution to Problem 8.8. (This covers the contents of [SM03, Thm. 9.2] with a different proof.)

What does this mean in practice? To solve a specific problem, one must

1. Pick a basis $q_0, \ldots, q_n$ of $\mathcal{P}_n$;
2. Compute M via $M_{k\ell} = \langle q_k, q_\ell \rangle$: this requires computing $\sim n^2$ integrals, once and for all for a given w—they can be stored;
3. Compute b via $b_k = \langle f, q_k \rangle$: this requires computing $n+1$ integrals, for each given f;
4. Solve the linear system $Mc = b$ (this can be simplified by preprocessing M, most likely via Cholesky factorization since $M \succ 0$—this can be stored instead of M.)

The solution is then $p_n(x) = c_0 q_0(x) + \cdots + c_n q_n(x)$, which is easy to evaluate if the basis polynomials are easy to evaluate. Computing the integrals is most often not doable by hand: in future lectures, we discuss how to compute them numerically. A more pressing problem is that M might be poorly conditioned (see example below.) We need to address that.

8.3 A geometric viewpoint

Best approximation in the 2-norm is (weighted) least squares approximation:

$$\text{Minimize: } \|f - p_n\|_2^2 = \int_a^b w(x)(f(x) - p_n(x))^2 \mathrm{d}x.$$

Just as in the finite dimensional case, the solution here also can be interpreted as the orthogonal projection of f to $\mathcal{P}_n$, where *orthogonal* is defined with

respect to the chosen inner product. To confirm equivalence, consider this chain of if and only if's:

$$\begin{aligned}
& p_n = \sum_{k=0}^{n} c_k q_k \text{ is the orthogonal projection of } f \text{ to } \mathcal{P}_n \\
\iff & f - p_n \text{ is orthogonal to } \mathcal{P}_n \\
\iff & \langle f - p_n, q \rangle = 0 \text{ for all } q \in \mathcal{P}_n \\
\iff & \langle f - p_n, q_\ell \rangle = 0 \text{ for } \ell = 0, \ldots, n \\
\iff & \langle f, q_\ell \rangle = \langle p_n, q_\ell \rangle \text{ for } \ell = 0, \ldots, n \\
\iff & b_\ell = \sum_{k=0}^{n} M_{\ell k} c_k \text{ for } \ell = 0, \ldots, n \\
\iff & b_\ell = (Mc)_\ell \text{ for } \ell = 0, \ldots, n \\
\iff & b = Mc.
\end{aligned}$$

This shows that there exists a unique orthogonal projection p_n of f to $\mathcal{P}_n$, and that the coefficients of p_n in the basis $q_0, \ldots, q_n$ are given by c: the unique solution to $Mc = b$. In the last section, we saw that $Mc = b$ also characterizes the best 2-norm approximation of f in $\mathcal{P}_n$, so that overall we find that, indeed, the orthogonal projection of f to $\mathcal{P}_n$ is the best 2-norm approximation of f in $\mathcal{P}_n$.

This last part is established with a different proof in the book. Specifically, using Cauchy–Schwarz in infinite dimension, it is first proved that the orthogonal projection is optimal. Then, it is proved that the optimum is unique by different means. Because it is often useful, we include a proof of Cauchy–Schwarz here.

Lemma 8.11 (Cauchy–Schwarz)**.** *For a given inner product space V, for any two vectors $f, g \in V$, it holds that*

$$|\langle f, g \rangle| \leq \|f\|_2 \|g\|_2.$$

Proof. Since $\|\cdot\|_2$ is a norm, for any $t \in \mathbb{R}$ we have:

$$0 \leq \|tf + g\|_2^2 = t^2 \|f\|_2^2 + 2t \langle f, g \rangle + \|g\|_2^2.$$

This holds in particular for $t = -\frac{\langle f,g \rangle}{\|f\|_2^2}$. (Notice that the right hand side is a convex quadratic in t: here, we choose t to be the minimizer of that quadratic, which yields the strongest possible inequality.) Plugging in this value of t we find:

$$0 \leq \frac{\langle f, g \rangle^2}{\|f\|_2^4} \|f\|_2^2 - 2 \frac{\langle f, g \rangle}{\|f\|_2^2} \langle f, g \rangle + \|g\|_2^2.$$

Reorganizing yields

$$\langle f, g\rangle^2 \leq \|f\|_2^2 \|g\|_2^2.$$

Take the square root to conclude. □

8.4 What could go wrong?

To fix ideas, let us pick $[a, b] = [0, 1]$. The most obvious choice for the weight function is $w(x) = 1$ for all x in $[0, 1]$. The most obvious choice of basis is the monomial basis: $q_k(x) = x^k$. In that case, the matrix M can be obtained analytically:

$$M_{k\ell} = \langle q_k, q_\ell \rangle = \int_0^1 x^k x^\ell \mathrm{d}x = \int_0^1 x^{k+\ell} \mathrm{d}x = \frac{x^{k+\ell+1}}{k+\ell+1}\Big|_0^1 = \frac{1}{k+\ell+1}.$$

This matrix is known as the Hilbert matrix, and it is plain terrible. The code below reveals that already for $n = 11$, the condition number is $\kappa(M) \approx 10^{16}$. In other words: upon solving $Mc = b$ in double precision, we cannot expect *any* digit to be correct for $n \geq 11$.

```
%% Hilbert matrix

% Matrix M corresponding to best 2-norm approximation with
% respect to the weight w(x) = 1 over the interval [0, 1] with
% the monomial basis 1, x, x^2, ..., x^n.
% Conditioning is terrible.

n = 11;
M = zeros(n+1, n+1);
for k = 0 : n
    for l = 0 : n
        M(k+1, l+1) = 1/(k+l+1);
    end
end
% Equivalently, M = hilb(n+1);

cond(M)
```

8.5 Orthogonal polynomials

The weight function w is typically tied to the application: we must consider it a given. Same goes for f, of course. Thus, our only lee-way is in the choice

of basis $q_0, \ldots, q_n$. We can espouse two different viewpoints which lead to the same conclusion:

> Algebraic viewpoint: Solving $Mc = b$ should be easy: choose q_k's such that M is diagonal;
>
> Geometric viewpoint: Orthogonal projections to $\mathcal{P}_n$ should be easy: choose q_k's to form an orthogonal basis.

Both considerations lead to the requirement $\langle q_k, q_\ell \rangle = 0$ for all $k \neq \ell$.[1]

One key point is: we do not want just $q_0, \ldots, q_n$ to form an orthogonal basis of $\mathcal{P}_n$: we want an infinite sequence of polynomials such that the first $n+1$ of them form such a basis for $\mathcal{P}_n$, for any n.

Definition 8.12 ([SM03, Def. 9.4]). *The sequence of polynomials $\phi_0, \phi_1, \phi_2, \ldots$ is a* system of orthogonal polynomials *on (a, b) with respect to the weight function w if*

1. *Each ϕ_k has degree exactly k,*
2. *$\langle \phi_k, \phi_\ell \rangle = 0$ if $k \neq \ell$, and*
3. *$\langle \phi_k, \phi_k \rangle \neq 0$.*

If the basis $q_0, \ldots, q_n$ is chosen as $\phi_0, \ldots, \phi_n$, solving the best approximation problem is straightforward since M is diagonal:

$$c_k = \frac{b_k}{M_{kk}} = \frac{\langle f, \phi_k \rangle}{\langle \phi_k, \phi_k \rangle}.$$

The denominator can be precomputed and stored. The numerator requires an integral which can be evaluated numerically (more on that later.) Furthermore, if the best approximation in $\mathcal{P}_n$ is unsatisfactory, then computing the best approximation in $\mathcal{P}_{n+1}$ only requires computing one additional coefficient, c_{n+1}. This is in stark contrast to previous approximation algorithms we have encountered.

This begs the question: how does one construct systems of orthogonal polynomials?

[1] For reasons that will become clear, we do not insist on forming an ortho*normal* basis for now.

8.5.1 Gram–Schmidt

Given a weight function w and a basis of polynomials $q_0, q_1, q_2, \ldots$ such that q_k has degree exactly k, we can apply the Gram–Schmidt procedure to figure out a system of orthogonal polynomials.[2] Concretely,

1. Let $\phi_0 = q_0$; then
2. Assuming $\phi_0, \ldots, \phi_n$ are constructed, build $\phi_{n+1} \in \mathcal{P}_{n+1}$ as:

$$\phi_{n+1} = q_{n+1} - a_n\phi_n - \cdots - a_0\phi_0,$$

where the coefficients $a_0, \ldots, a_n$ must be chosen so that $\langle \phi_{n+1}, \phi_k \rangle = 0$ for $k = 0, \ldots, n$. That is,

$$0 = \langle \phi_{n+1}, \phi_k \rangle = \langle q_{n+1}, \phi_k \rangle - a_k \langle \phi_k, \phi_k \rangle .$$

Hence,

$$a_k = \frac{\langle q_{n+1}, \phi_k \rangle}{\langle \phi_k, \phi_k \rangle}.$$

Computing the coefficients a_k involves integrals which may or may not be computable by hand. Even if this is figured out, a more important issue remains: evaluating the polynomial ϕ_n at some point x requires $\sim n^2$ flops on top of evaluating $q_0(x), \ldots, q_n(x)$ (think about it.) This is expensive. We can do better.

Question 8.13. *Orthogonalize the monomial basis $1, x, x^2, x^3, \ldots$ with respect to the weight $w(x) = 1$ over $[-1, 1]$. The resulting system of polynomials (upon normalizing such that $\phi_k(1) = 1$ for all k) is known as the* Legendre polynomials. *(Just compute the first few polynomials to get the idea.)*

■

8.5.2 A look at the Chebyshev polynomials

We actually already encountered a system of orthogonal polynomials. Indeed, the Chebyshev polynomials (Definition 7.10) are orthogonal on $[-1, 1]$

[2] We won't use Gram–Schmidt in practice, so there is no need to worry about numerical stability in the face of inexact arithmetic: classical Gram–Schmidt will do.

with respect to the weight $w(x) = \frac{1}{\sqrt{1-x^2}}$. This weight function puts more emphasis on accurate approximation near ± 1.[3]

To verify orthogonality, recall

$$T_n(x) = \cos(n \arccos(x))$$

for $x \in [-1, 1]$, so that

$$\begin{aligned}
M_{k,\ell} = \langle T_k, T_\ell \rangle &= \int_{-1}^{1} T_k(x) T_\ell(x) w(x) \mathrm{d}x \\
&= \int_{-1}^{1} \cos(k \arccos(x)) \cos(\ell \arccos(x)) \frac{1}{\sqrt{1-x^2}} \mathrm{d}x \\
\text{(change: } x = \cos\theta) \quad &= \int_{\pi}^{0} \cos(k\theta) \cos(\ell\theta) \frac{1}{|\sin\theta|} (-\sin\theta) \mathrm{d}\theta \\
&= \int_{0}^{\pi} \cos(k\theta) \cos(\ell\theta) \mathrm{d}\theta \\
&= \frac{1}{2} \int_{0}^{\pi} \cos((k-\ell)\theta) + \cos((k+\ell)\theta) \mathrm{d}\theta \\
&= \begin{cases} 0 & \text{if } k \neq \ell, \\ \pi & \text{if } k = \ell = 0, \\ \frac{\pi}{2} & \text{if } k = \ell \neq 0. \end{cases}
\end{aligned}$$

This specifies M entirely. Furthermore,

$$b_k = \langle f, T_k \rangle = \int_{-1}^{1} f(x) \cos(k \arccos(x)) \frac{\mathrm{d}x}{\sqrt{1-x^2}} = \int_{0}^{\pi} f(\cos\theta) \cos(k\theta) \mathrm{d}\theta.$$

Hence,

$$p_n = c_0 T_0 + \cdots + c_n T_n$$

is the best 2-norm approximation to f over $[-1, 1]$ with respect to the weight $w(x) = \frac{1}{\sqrt{1-x^2}}$ if and only if

$$\begin{aligned}
c_0 &= \frac{b_0}{M_{00}} = \frac{1}{\pi} \int_{0}^{\pi} f(\cos\theta) \mathrm{d}\theta, \\
c_k &= \frac{b_k}{M_{kk}} = \frac{2}{\pi} \int_{0}^{\pi} f(\cos\theta) \cos(k\theta) \mathrm{d}\theta, \qquad \text{for } k = 1, 2, \ldots
\end{aligned}$$

(This is closely related to taking a Fourier transform of $f \circ \cos$.)

[3] It is now clear why we needed to allow w to take on infinite values at the extreme points of the interval.

8.5.3 Three-term recurrence relations

The Chebyshev polynomials are orthogonal, *and* they obey a three-term recurrence relation. This is not an accident. We now show that all systems of orthogonal polynomials obey such a recurrence.

This notably means that it is only necessary to figure out the coefficients of this recurrence to obtain a cheap and reliable way of computing the polynomials. This is highly preferable to the Gram–Schmidt procedure. The theorem below gives explicit formulas for these coefficients that can be evaluated in practice for a given weight w. For various important weights, the recurrence coefficients can also be looked up in a book.

Theorem 8.14. *Let $\phi_0, \phi_1, \phi_2 \ldots$ be a system of orthogonal polynomials with respect to some weight w on (a, b) and such that each polynomial is* monic*, that is, $\phi_k(x) = x^k +$ lower order terms (the leading coefficient is 1.) Then,*

$$\begin{aligned} \phi_0(x) &= 1, \\ \phi_1(x) &= x - \alpha_0, \\ \phi_{k+1}(x) &= (x - \alpha_k)\phi_k(x) - \beta_k^2 \phi_{k-1}(x), \qquad \text{for } k = 1, 2, \ldots \end{aligned} \tag{8.6}$$

and

$$\begin{aligned} \alpha_k &= \frac{\langle x\phi_k, \phi_k \rangle}{\|\phi_k\|_2^2}, && \text{for } k = 0, 1, 2, \ldots \\ \beta_k^2 &= \frac{\|\phi_k\|_2^2}{\|\phi_{k-1}\|_2^2}, && \text{for } k = 1, 2, \ldots \end{aligned}$$

Before we get to the proof, a couple remarks:

1. If the coefficients α_k, β_k^2 are known (precomputed and stored), then evaluating $\phi_0, \ldots, \phi_n$ at a given x requires $\sim 4n$ flops. (Check it: you first compute $\phi_0(x)$ and $\phi_1(x)$ (for 0 and 1 flop respectively), then unroll the recurrence to successively compute $\phi_2(x), \ldots, \phi_n(x)$ for 4 flops each.)

2. This theorem also shows the system of orthogonal polynomials with respect to w is unique up to scaling—the scaling is fixed by requiring them to be monic.

Proof. $\phi_0(x) = 1$ is the only monic polynomial of degree 0, hence this is the only possibility. The polynomial $\phi_1(x)$ must be of the form $x - \alpha_0$ for some α_0: imposing $0 = \langle \phi_1, \phi_0 \rangle = \langle x - \alpha_0, 1 \rangle$ gives $\alpha_0 = \frac{\langle x,1 \rangle}{\langle 1,1 \rangle}$ as prescribed.

Now consider $k \geq 1$. Since ϕ_k's are monic,

$$\begin{aligned}\phi_k(x) &= x^k + \text{“some polynomial of degree } \leq k-1\text{”, and}\\ \phi_{k+1}(x) &= x^{k+1} + \text{“some polynomial of degree } \leq k\text{”.}\end{aligned}$$

Hence, $x\phi_k - \phi_{k+1}$ is a polynomial of degree at most k and we can write:

$$x\phi_k - \phi_{k+1} = c_0\phi_0 + \cdots + c_k\phi_k. \tag{8.7}$$

Our primary goal is to show that only c_k and c_{k-1} are nonzero. To this end, first take inner products of (8.7) with ϕ_ℓ for some ℓ:

$$\langle x\phi_k, \phi_\ell\rangle - \langle \phi_{k+1}, \phi_\ell\rangle = \sum_{j=0}^{k} c_j \langle \phi_j, \phi_\ell\rangle .$$

For $\ell \leq k$, orthogonality leads to:

$$\langle x\phi_k, \phi_\ell\rangle = c_\ell \langle \phi_\ell, \phi_\ell\rangle . \tag{8.8}$$

For $\ell = k$, we find

$$c_k = \frac{\langle x\phi_k, \phi_k\rangle}{\|\phi_k\|_2^2} = \alpha_k. \tag{8.9}$$

For $\ell = k-1$, equation (8.8) gives:

$$c_{k-1} = \frac{\langle x\phi_k, \phi_{k-1}\rangle}{\|\phi_{k-1}\|_2^2}.$$

We can simplify this further, using a very special property:

$$\langle x\phi_k, \phi_{k-1}\rangle = \int_a^b x\phi_k(x)\phi_{k-1}(x)w(x)\mathrm{d}x = \langle \phi_k, x\phi_{k-1}\rangle .$$

Think about this last equation: it's rather special. We exploit it as follows:

$$\begin{aligned}x\phi_{k-1} &= x^k + \text{“some polynomial of degree } \leq k-1\text{”}\\ &= \phi_k + \underbrace{\text{“some other polynomial of degree } \leq k-1\text{”}}_{q\in\mathcal{P}_{k-1}}.\end{aligned}$$

We can expand q in the basis $\phi_0, \ldots, \phi_{k-1}$:

$$q = a_0\phi_0 + \cdots + a_{k-1}\phi_{k-1}.$$

Since ϕ_k is orthogonal to $\phi_0, \ldots, \phi_{k-1}$, it follows that ϕ_k is orthogonal to q as well[4] and

$$\langle x\phi_k, \phi_{k-1}\rangle = \langle \phi_k, x\phi_{k-1}\rangle = \langle \phi_k, \phi_k + q\rangle = \langle \phi_k, \phi_k\rangle .$$

Thus,

$$c_{k-1} = \frac{\|\phi_k\|_2^2}{\|\phi_{k-1}\|_2^2} = \beta_k^2. \tag{8.10}$$

Finally, consider (8.8) for $\ell \leq k-2$. Then,

$$\langle x\phi_k, \phi_\ell\rangle = \langle \phi_k, x\phi_\ell\rangle = 0,$$

since $x\phi_\ell$ is a polynomial of degree $\ell + 1 \leq k-1$: it is orthogonal to ϕ_k by the same argument as above. Hence,

$$c_0 = \cdots = c_{k-2} = 0. \tag{8.11}$$

Collect the numbered equations to conclude. □

Remark 8.15. *A converse of this theorem exists, see the Shohat–Favard theorem* https://en.wikipedia.org/wiki/Favard%27s_theorem.

Question 8.16. *Assume w is an even weight function over the interval $[-1, 1]$. In the notation of the above theorem, show ϕ_k is even if k is even and ϕ_k is odd if k is odd, and show the coefficients α_k are all zero. (The two are best shown together, as one implies the other, in alternation.)*

■

Question 8.17. *Using the expression $T_n(x) = \cos(n \arccos(x))$ for Chebyshev polynomials and the fact they are orthogonal with respect to $w(x) = \frac{1}{\sqrt{1-x^2}}$, use the theorem above to recover the recurrence relation for the T_n's. (Be mindful of normalization: T_n is not monic.)*

■

Remark 8.18. *For Legendre polynomials ($w(x) = 1, x \in [-1, 1]$), explicit formulas are known:*

$$\alpha_k = 0, \qquad \beta_k^2 = \frac{k^2}{4k^2 - 1}.$$

[4] That is: ϕ_k is orthogonal to $\mathcal{P}_{k-1}$. This is important. We're going to use this again.

Likewise, for Chebyshev polynomials:

$$\alpha_k = 0, \qquad \beta_k^2 = \begin{cases} \frac{1}{2} & \text{if } k = 1, \\ \frac{1}{4} & \text{if } k \geq 2. \end{cases}$$

In both cases, the recurrence provides the orthogonal polynomials scaled to be monic. Notice how these coefficients are quite close (consider the asymptotics for $k \to \infty$.)

8.5.4 Roots of orthogonal polynomials

Polynomials orthogonal for an inner product over (a, b) have distinct roots in (a, b).

Theorem 8.19 (Thm. 9.4 in [SM03])**.** *Let $\phi_0, \phi_1, \phi_2, \ldots$ form a system of orthogonal polynomials with respect to the weight w on (a, b). For $j \geq 1$, the j roots of ϕ_j are real and distinct and lie in (a, b).*

Proof. Assume $\xi_1, \ldots, \xi_k$ are the k points in (a, b) where ϕ_j changes sign.[5] There is at least one such point. Indeed, ϕ_j is orthogonal to $\phi_0(x) = 1$ so that

$$0 = \langle \phi_j, \phi_0 \rangle = \int_a^b \phi_j(x) w(x) \mathrm{d}x,$$

which implies ϕ_j changes sign at least once on (a, b). (Here, we used that ϕ_j is not identically zero, that w is positive, and that both are continuous on (a, b).) Define

$$\pi_k(x) = (x - \xi_1) \cdots (x - \xi_k).$$

Then, the product $\phi_j(x)\pi_k(x)$ no longer changes sign in (a, b). This implies

$$0 \neq \int_a^b \phi_j(x) \pi_k(x) w(x) \mathrm{d}x = \langle \phi_j, \pi_k \rangle .$$

Now, π_k is a polynomial of degree k, and ϕ_j is orthogonal to all polynomials of degree up to $j - 1$, so that it must be that π_k has degree at least j: $k \geq j$. On the other hand, ϕ_j cannot change sign more than j times since it has degree j. Thus, $k = j$, as desired: the points $\xi_1, \ldots, \xi_j$ are (all) the roots of ϕ_j, distinct in (a, b). □

[5] Thus, double roots, quadruple roots etc. do not count.

In the chapters about integration, we will see that the roots of orthogonal polynomials are particularly appropriate to design numerical integration schemes (Gauss quadrature rules.) How can we compute these roots? As it turns out, they are the eigenvalues of a tridiagonal matrix.[6] This means any of our fast and reliable algorithms to compute eigenvalues of tridiagonal matrices can be used here. But more on this in the next chapter, about integration.

8.5.5 Differential equations & orthogonal polynomials

If you go to the Wikipedia pages for Legendre polynomials[7] and Chebyshev polynomials,[8] you will find that the defining property put forward for both of these is that they are the solutions to a certain differential equation. We took a very different route, but evidently that perspective is important enough to take center stage in other contexts. This side comment aims to give a high level sense of how these equations are related to our polynomials.

You certainly know the *spectral theorem*: it says that a symmetric matrix $A \in \mathbb{R}^{n\times n}$ admits n real eigenvalues, and that associated eigenvectors form an orthogonal basis for $\mathbb{R}^n$. We can think of $\mathbb{R}^n$ as a vector space (of finite dimension), and choose the standard inner product $\langle x, y\rangle = x^T y$. And we can think of A as a linear operator: $A\colon \mathbb{R}^n \to \mathbb{R}^n$. A matrix is symmetric if $A = A^T$. You can check that this is equivalent to stating that $\langle x, Ay\rangle = \langle Ax, y\rangle$ for all $x, y \in \mathbb{R}^n$. An operator with the latter property is called self-adjoint.

There exists an equivalent of the spectral theorem for (compact) linear operators on *Hilbert spaces*, that is, inner product spaces (possibly infinite dimensional) that are complete for the metric induced by the chosen inner product.[9] For example, $L^2_w(a, b)$ is a Hilbert space for the inner products we have encountered here. A linear operator $A\colon L^2_w(a, b) \to L^2_w(a, b)$ maps functions to functions in that space. For example, A could be a differential operator (differentiating a function, under some conditions, gives another function.) There exists a notion of eigenvalue and corresponding eigenfunction (the equivalent of an eigenvector) for A: solutions of $Af = \lambda f$. If A is self-adjoint (in the sense that $\langle Af, g\rangle = \langle f, Ag\rangle$, where f, g are two functions) and if it is compact, then the spectral theorem says λ's are real and, crucially, the eigenfunctions are orthogonal for the chosen inner product.

[6] This shouldn't be too surprising; In our work about Sturm sequences, we reduced the problem of computing eigenvalues of tridiagonal matrices to that of computing the roots of polynomials expressed via a three-term recurrence relation.

[7] https://en.wikipedia.org/wiki/Legendre_polynomials

[8] https://en.wikipedia.org/wiki/Chebyshev_polynomials

[9] https://en.wikipedia.org/wiki/Compact_operator_on_Hilbert_space

How is this related to Legendre and Chebyshev polynomials? Take another look at these differential equations that appear at the top of their Wikipedia pages: they are of the form "some differential operator applied to f = some multiple of f". In other words: the orthogonal polynomials are the eigenfunctions of that particular (compact) linear differential operator on a Hilbert space of functions, for the corresponding inner product (that is, the choice of weight function w on (a, b)).

This is all part of a more general theory of Sturm–Liouville operators,[10] which provide a way to produce orthogonal systems of eigenfunctions on a large class of Hilbert spaces. The tip of a beautiful iceberg.

[10] https://en.wikipedia.org/wiki/Sturm%E2%80%93Liouville_theory

Chapter 9

Integration

We consider the problem of computing integrals.

Problem 9.1. *Given* $f \in C[a,b]$, *compute* $\int_a^b f(x)\mathrm{d}x$.

For most functions f, this integral cannot be computed analytically. Think for example of $f(x) = e^{-x^2}$ (whose integrals come up frequently in probability computations involving Gaussians), of $f(x) = \log(2 + \cos(5e^x - \sin(x)))$, and of cases where $f(x)$ is the result of a complex computation, that may not even be known to us (black box model.) Thus, we resort to numerical algorithms to approximate it.

Informed by the previous chapters, our main strategy is to approximate f with a polynomial p_n in $\mathcal{P}_n$, and to integrate p_n instead of f. Presumably, if $f \approx p_n$, then $\int_a^b f(x)\mathrm{d}x \approx \int_a^b p_n(x)\mathrm{d}x$. Integrating the polynomial should pose no difficulty. To legitimize this intuition, an important aspect is to give a precise, quantified meaning to these "$\approx$" signs.

We discussed three ways to approximate a function with a polynomial: interpolation, minimax, and best in the 2-norm. For minimax, we did not discuss concrete algorithms (and existing ones are involved: we don't want to work that hard to answer our simple question.) For 2-norm approximations, one actually needs to compute integrals numerically: that would be circular. This leaves interpolation.

Recall Lagrange interpolation. We only need to make a choice of interpolation points $a \leq x_0 < \cdots < x_n \leq b$. Then,

$$f(x) \approx p_n(x) = \sum_{k=0}^{n} f(x_k) L_k(x), \qquad L_k(x) = \prod_{i \neq k} \frac{x - x_i}{x_k - x_i}.$$

Then, informally,

$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b p_n(x)\mathrm{d}x = \sum_{k=0}^{n} f(x_k) \underbrace{\int_a^b L_k(x)\mathrm{d}x}_{w_k} = \sum_{k=0}^{n} w_k f(x_k).$$

Some work goes into computing the *quadrature weights* $w_0, \ldots, w_n$—but notice that this is independent of f: it needs only be done once, and the weights can be stored to disk. Then, the *quadrature rule* $\sum_{k=0}^{n} w_k f(x_k)$ is easily applied. The points $x_0, \ldots, x_n$ where f needs to be evaluated are called *quadrature points*.

9.1 Computing the weights

The weights

$$w_k = \int_a^b L_k(x)\mathrm{d}x$$

depend on the quadrature points $x_0, \ldots, x_n$. The formula suggests an obvious way to compute the weights. For the sake of example, let us consider the case where the quadrature points are equispaced on $[a, b]$, giving so-called *Newton–Cotes quadrature rules*.

Trapezium rule (Newton–Cotes with $n = 1$). With $x_0 = a, x_1 = b$, the Lagrange polynomials have degree 1:

$$L_0(x) = \frac{x-b}{a-b}, \qquad L_1(x) = \frac{x-a}{b-a}.$$

These are easily integrated by hand:

$$w_0 = \int_a^b \frac{x-b}{a-b}\mathrm{d}x = \frac{b-a}{2}, \qquad w_1 = \int_a^b \frac{x-a}{b-a}\mathrm{d}x = \frac{b-a}{2}.$$

Overall, we get the quadrature rule

$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b p_1(x)\mathrm{d}x = \frac{b-a}{2}(f(a)+f(b)).$$

This is the area of the trapezium defined by the points $(a, 0)$, $(a, f(a))$, $(b, f(b))$, $(b, 0)$. Notice the symmetry here: it makes sense that $f(a)$ and $f(b)$ get the same weight.

Simpson's rule (Newton–Cotes with $n = 2$). With equispaced points $x_0 = a, x_1 = \frac{a+b}{2}, x_2 = b$, Lagrange polynomials have degree 2. We could obtain the weights exactly as above, but this is rapidly becoming tedious. Let's keep an eye out for shortcuts as we proceed. Consider w_0 first:

$$w_0 = \int_a^b L_0(x)\mathrm{d}x = \int_a^b \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \text{...some work...} = \frac{b-a}{6}.$$

Similar computations yield w_1, w_2. But we can save some work here. For example, by a symmetry argument similar as above, we expect $w_2 = w_0$. You can verify it. What about w_1? Let's think about the (very) special case $f(x) = 1$. Since f is a polynomial of degree 0, its interpolation polynomial in $\mathcal{P}_2$ is exact: $f = p_2$. Thus,

$$\begin{aligned} b-a &= \int_a^b f(x)\mathrm{d}x \\ &= \int_a^b p_2(x)\mathrm{d}x = w_0 f(x_0) + w_1 f(x_1) + w_2 f(x_2) = w_0 + w_1 + w_2. \end{aligned}$$

That is: the weights sum to $b-a$ (the length of the interval.) This allows to conclude already:

$$w_0 = \frac{b-a}{6}, \qquad w_1 = 4\frac{b-a}{6}, \qquad w_2 = \frac{b-a}{6}.$$

This rule interpolates f at three points with a quadratic and approximates the integral of f with the integral of the quadratic.

Let's revisit the argument above: $f(x) = 1$ is integrated exactly upon approximation by p_2, because $p_2 = f$. More generally,

If $f \in \mathcal{P}_n$, then $f = p_n$.
Hence, if $f \in \mathcal{P}_n$, then $\int_a^b f(x)\mathrm{d}x = \int_a^b p_n(x)\mathrm{d}x = \sum_{k=0}^n w_k f(x_k)$.

This last point is crucial.

We can use this to our advantage to automate the computation of weights w_k. Indeed, with $n+1$ points, the integration rule is exact for polynomials of degree up to n, which, equivalently, means it is exact for polynomials $1, x, x^2, \ldots, x^n$. Each of these yields a linear equation in the w_k's. Consider $f(x) = x^i$ for $i = 0, \ldots, n$:

$$\frac{b^{i+1} - a^{i+1}}{i+1} = \int_a^b x^i \mathrm{d}x = \sum_{k=0}^n w_k x_k^i.$$

These $n+1$ linear equations can be arranged in matrix form:

$$\begin{bmatrix} x_0^0 & \cdots & x_n^0 \\ \vdots & & \vdots \\ x_0^n & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} b-a \\ \vdots \\ \frac{b^{n+1}-a^{n+1}}{n+1} \end{bmatrix}. \tag{9.1}$$

We recognize an old friend: the matrix is a (transposed) Vandermonde matrix. We know from experience that it is often ill conditioned even for n below 100. We will circumvent this for more sophisticated methods later. Still, this system is easily set up for the general case (general n, and general choice of quadrature points) and can be solved easily for small n. For larger n, recall that we only need to solve the system *once* accurately. This could be done offline, for example using variable precision arithmetic packages (`vpa` in Matlab's symbolic toolbox, for example), then rounding the weights to double precision. This of course also has its practical limitations. For these reasons, we put conditioning concerns aside for now and proceed, bearing in mind that it may be unwise to aim for very large values of n, unless the w_k's are computed in a better way.[1]

9.2 Bounding the error

Our goal now is to bound the truncation error:

$$E_n(f) = \left| \int_a^b f(x)\mathrm{d}x - \sum_{k=0}^{n} w_k f(x_k) \right| = \left| \int_a^b f(x)\mathrm{d}x - \int_a^b p_n(x)\mathrm{d}x \right|.$$

Recall Theorem 6.5: if $f \in C^{n+1}[a,b]$ (meaning: if f is $n+1$ times continuously differentiable on $[a,b]$) and $p_n \in \mathcal{P}_n$ interpolates f at $a \le x_0 < \ldots < x_n \le b$, then

$$|f(x) - p_n(x)| \le \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} |\pi_{n+1}(x)|,$$

[1] In previous chapters, to address ill-conditioning of the Vandermonde matrix, we changed bases: instead of using monomials $1, x, x^2, \ldots$ we used Lagrange polynomials $L_0, L_1, L_2, \ldots$ which turned the Vandermonde matrix into an identity matrix. Notice that doing this here, that is, imposing that $L_0, \ldots, L_n$ are integrated exactly, only shifts the burden to that of computing the right hand side, which contains $\int_a^b L_k(x)\mathrm{d}x$: this is our original task. At any rate, later in this chapter we will see that there are better ways to build quadrature rules, namely, composite rules and Gaussian rules.

where $\pi_{n+1}(x) = (x - x_0) \cdots (x - x_n)$. This leads to a simple bound:

$$\begin{aligned} E_n(f) &= \left| \int_a^b f(x) - p_n(x) \mathrm{d}x \right| \\ &\leq \int_a^b |f(x) - p_n(x)| \mathrm{d}x \\ &\leq \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} \int_a^b |\pi_{n+1}(x)| \mathrm{d}x \qquad (9.2) \\ &\leq \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} (b-a) \|\pi_{n+1}\|_\infty. \qquad (9.3) \end{aligned}$$

Both (9.2) and (9.3) can be used to obtain an explicit bound on $E_n(f)$. Using (9.3) is easier given our previous work so we will use this one. See [SM03, §7] for an explicit use of (9.2), which gives slightly better constants.

As a general observation, unsurprisingly, if the quadrature nodes $x_0 < \cdots < x_n$ are chosen so as to make $\|\pi_{n+1}\|_\infty$ small (a choice independent of f), then so is the integration error, provided $f^{(n+1)}$ is defined, continuous and not too large. This echoes our considerations from the chapter about interpolation. Thus, we also expect that it is better to use Chebyshev nodes rather than equispaced nodes as quadrature points.

Equispaced points (Newton–Cotes). Recall

$$\|\pi_{n+1}\|_\infty \leq \frac{n! \left(\frac{b-a}{n}\right)^{n+1}}{4}.$$

As a result,

$$E_n(f) \leq \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+2}}{4(n+1)n^{n+1}}. \qquad \text{(Newton–Cotes)}$$

This gives in particular:

$$E_1(f) \leq \|f''\|_\infty \frac{(b-a)^3}{8}, \qquad \text{(trapezium rule)}$$

$$E_2(f) \leq \|f'''\|_\infty \frac{(b-a)^4}{96}. \qquad \text{(Simpson's rule)}$$

For small n, we can get an exact expression for $\|\pi_n\|_\infty$. For $n = 1$, it is easy to derive: $\|\pi_2\|_\infty = \frac{(b-a)^2}{4}$. For $n = 2$, it takes a bit of work (with the help of Wolfram for example, as otherwise it is quite tedious), and we get $\|\pi_3\|_\infty = \frac{(b-a)^3}{12\sqrt{3}}$. This is not too important: it only changes constants somewhat.

Chebyshev points. Recall equations (7.7) and (7.8): for $n \geq 0$, choosing Chebyshev nodes on $[-1, 1]$ as in (7.9) leads to:

$$\pi_{n+1}(x) = (x - x_0) \cdots (x - x_n) = \frac{1}{2^n} T_{n+1}(x),$$

whose infinity norm is $1/2^n$. What about a general interval $[a, b]$? Consider the linear change of variable

$$t = t(x) = \frac{a+b}{2} + \frac{b-a}{2}x,$$

constructed to satisfy $t(-1) = a$ and $t(1) = b$: it maps $[-1, 1]$ to $[a, b]$. The Chebyshev nodes on $[a, b]$ are defined as

$$t_k = t(x_k), \qquad \text{for } k = 0, \ldots, n.$$

Interpolation at those points leads to a different π_{n+1} polynomial:

$$\begin{aligned}\tilde{\pi}_{n+1}(t) &\triangleq (t - t_0) \cdots (t - t_n) \\ &= \left(t - \frac{a+b}{2} - \frac{b-a}{2}x_0\right) \cdots \left(t - \frac{a+b}{2} - \frac{b-a}{2}x_n\right).\end{aligned}$$

Consider the inverse of the change of variable:

$$x = x(t) = \frac{2}{b-a}\left(t - \frac{a+b}{2}\right).$$

Plugging this into the expression for $\tilde{\pi}_{n+1}(t)$ and factoring out $\frac{b-a}{2}$ from each of the $n+1$ terms gives:

$$\tilde{\pi}_{n+1}(t) = \left(\frac{b-a}{2}\right)^{n+1} (x - x_0) \cdots (x - x_n) = \left(\frac{b-a}{2}\right)^{n+1} \pi_{n+1}(x).$$

As a result,

$$\|\tilde{\pi}_{n+1}\|_\infty = \left(\frac{b-a}{2}\right)^{n+1} \frac{1}{2^n}.$$

Combining with (9.3) gives:

$$E_n(f) \leq \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+2}}{(n+1)!\, 2^{2n+1}}. \qquad \text{(Chebyshev nodes)} \qquad (9.4)$$

In particular,

$$\begin{aligned}E_1(f) &\leq \|f''\|_\infty \frac{(b-a)^3}{16}, \\ E_2(f) &\leq \|f'''\|_\infty \frac{(b-a)^4}{192}.\end{aligned}$$

9.3 Composite rules

We covered composite rules in class. See [SM03, §7.5], including error bounds (you can use the error bounds derived above instead, which gives $\sim \frac{1}{m^3}$ for composite Simpson's instead of $\sim \frac{1}{m^4}$.)

Notice why it is important to work with Newton–Cotes rules here: they allow to reuse extreme points of subintervals, instead of needing to recompute them.

Composite rules are quite important in practice: there are no typeset notes here because the book is sufficiently explicit; do not take it as a sign that this is less important :).

9.4 Gaussian quadratures

The quadrature rules discussed above, based on interpolation at $n+1$ quadrature points, integrate polynomials in $\mathcal{P}_n$ exactly.

Definition 9.2. *A quadrature rule has* degree of precision n *if it integrates all polynomials in $\mathcal{P}_n$ exactly, but not so for $\mathcal{P}_{n+1}$.*

Thus, the rules presented above have degree of precision at least n.

Question 9.3. *Show that Newton–Cotes with $n = 1$ has degree of precision 1, yet Newton–Cotes with $n = 2$ has degree of precision 3. More generally, reason that Newton–Cotes rules have degree of precision n if n is odd, and degree of precision $n + 1$ if n is even.*

■

The notion of degree of precision of a rule suggests a natural question:

> *Using $n + 1$ quadrature points, what is the highest degree of precision possible for a quadrature rule?*

Let's put a limit on our dreams. For a given quadrature rule with quadrature nodes $x_0, \ldots, x_n$ and weights $w_0, \ldots, w_n$, consider the following polynomial:

$$q(x) = (x - x_0)^2 \cdots (x - x_n)^2.$$

This is a polynomial of degree $2n + 2$. Surely, $\int_a^b q(x)\mathrm{d}x$ is positive (in particular, nonzero). Yet, the quadrature rule yields $\sum_{k=0}^{n} w_k q(x_k) = 0$, since all the quadrature points are roots of q. We conclude that no quadrature rule based on $n+1$ points can integrate q correctly; in other words: no quadrature

rule based on $n+1$ points has degree of precision $2n+2$ or more. Fine, we do not expect to find a method with degree of precision more than $2n+1$. But can we attain this high degree?

The answer, magically to some extent, is yes! It goes by the name of *Gauss quadrature*, and involves orthogonal polynomials. Before we get into it, let us extend the scope of the integration problem somewhat, to allow for weights $w(x)$ (this is notably useful for best approximation in the 2-norm):

Problem 9.4. *Given $f \in C[a,b]$ and a weight function w on (a,b), compute $\int_a^b f(x)w(x)\mathrm{d}x$.*

Consider a system of orthogonal polynomials for the inner product with weight w,

$$\phi_0, \phi_1, \phi_2, \ldots,$$

as defined in Definition 8.12. Observe that for any polynomial $p_{2n+1} \in \mathcal{P}_{2n+1}$, there exist two polynomials q and r in $\mathcal{P}_n$ such that

$$p_{2n+1}(x) = q(x)\phi_{n+1}(x) + r(x).$$

(That is, q is the quotient and r is the remainder after division of p_{2n+1} by ϕ_{n+1}.) Furthermore, let $x_0 < \cdots < x_n$ in $[a,b]$ and $w_0, \ldots, w_n$ form a quadrature rule of degree of precision at least n (to be determined.) Then,

$$\begin{aligned}
\int_a^b p_{2n+1}(x)w(x)\mathrm{d}x &= \int_a^b (q(x)\phi_{n+1}(x) + r(x))w(x)\mathrm{d}x \\
&= \underbrace{\int_a^b q(x)\phi_{n+1}(x)w(x)\mathrm{d}x}_{=\langle q,\phi_{n+1}\rangle=0 \text{ since } q\in\mathcal{P}_n} + \int_a^b r(x)w(x)\mathrm{d}x \\
&= \sum_{k=0}^{n} w_k r(x_k) \qquad \text{(since the rule is exact for } r \in \mathcal{P}_n.)
\end{aligned}$$

Thus, using the quadrature rule of degree of precision at least n, one could conceivably integrate all polynomials of degree up to $2n+1$, if only one could evaluate r instead of p_{2n+1} at the quadrature nodes. In general, this is not an easy task. Here comes the key part:

> *If we pick the quadrature nodes $x_0 < \ldots < x_n$ to be the $n+1$ roots of ϕ_{n+1} (known to be real, distinct and in $[a,b]$), then*
>
> $$p_{2n+1}(x_k) = q(x_k)\phi_{n+1}(x_k) + r(x_k) = r(x_k).$$

Consequently,

$$\int_a^b p_{2n+1}(x)w(x)\mathrm{d}x = \sum_{k=0}^{n} w_k p_{2n+1}(x_k).$$

The weights w_k can be determined as usual from (9.1)—but we will do better.

Thus, the roots of orthogonal polynomials tell us where to sample f in order to maximize the degree of precision of the quadrature rule. How do we compute those roots?

9.4.1 Computing roots of orthogonal polynomials

Recall Theorem 8.19, which states that roots of orthogonal polynomials are real and distinct and lie in (a, b). Now, we need to compute them. We will show they are the eigenvalues of a symmetric, tridiagonal matrix: we know how to handle that. We only need to find this tridiagonal matrix. First, we will find a non-symmetric tridiagonal matrix; then, we will show it has the same eigenvalues as a symmetric one.

It all hinges on the three-term recurrence, Theorem 8.14.[2] As a matter of example, let us aim to find the roots of ϕ_5. The three-term recurrence gives:

$$\begin{aligned}
\alpha_0\phi_0(x) + \phi_1(x) &= x\phi_0(x)\\
\beta_1^2\phi_0(x) + \alpha_1\phi_1(x) + \phi_2(x) &= x\phi_1(x)\\
\beta_2^2\phi_1(x) + \alpha_2\phi_2(x) + \phi_3(x) &= x\phi_2(x)\\
\beta_3^2\phi_2(x) + \alpha_3\phi_3(x) + \phi_4(x) &= x\phi_3(x)\\
\beta_4^2\phi_3(x) + \alpha_4\phi_4(x) \underbrace{+ \phi_5(x)}_{\to\text{rhs}} &= x\phi_4(x).
\end{aligned}$$

In matrix form, this reads:

$$\underbrace{\begin{bmatrix}
\alpha_0 & 1 & & & \\
\beta_1^2 & \alpha_1 & 1 & & \\
 & \beta_2^2 & \alpha_2 & 1 & \\
 & & \beta_3^2 & \alpha_3 & 1\\
 & & & \beta_4^2 & \alpha_4
\end{bmatrix}}_{J}
\underbrace{\begin{bmatrix}\phi_0(x)\\ \phi_1(x)\\ \phi_2(x)\\ \phi_3(x)\\ \phi_4(x)\end{bmatrix}}_{v(x)}
= x\underbrace{\begin{bmatrix}\phi_0(x)\\ \phi_1(x)\\ \phi_2(x)\\ \phi_3(x)\\ \phi_4(x)\end{bmatrix}}_{v(x)}
- \begin{bmatrix}0\\0\\0\\0\\ \phi_5(x)\end{bmatrix}.$$

The matrix on the left, J, is the *Jacobi matrix*. The crucial observation follows:

[2] The recurrence is set up assuming the polynomials are monic, which doesn't affect their roots so this is inconsequential to our endeavor.

If x is a root of ϕ_5, then

$$Jv(x) = xv(x). \tag{9.5}$$

That is: if x is a root of ϕ_5, then x is an eigenvalue of the Jacobi matrix, with eigenvector $v(x)$.[3]

Let's say this again: *all five distinct roots of ϕ_5 are eigenvalues of J.* Since J is a 5×5 matrix, it has five eigenvalues, so that *the roots of ϕ_5 are exactly the eigenvalues of J.* This generalizes for all n of course.

Theorem 9.5. *The roots of ϕ_{n+1} are the eigenvalues of the Jacobi matrix*

$$J_{n+1} = \begin{bmatrix} \alpha_0 & 1 & & & \\ \beta_1^2 & \alpha_1 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1}^2 & \alpha_{n-1} & 1 \\ & & & \beta_n^2 & \alpha_n \end{bmatrix}.$$

This matrix is tridiagonal, but it is not symmetric. The eigenvalue computation algorithms we have discussed require a symmetric matrix. Let's try to make J symmetric without changing its eigenvalues. Using a diagonal similarity transformation (coefficients $s_k \neq 0$ to be determined):

$$\overbrace{\begin{bmatrix} s_0^{-1} & & & & \\ & s_1^{-1} & & & \\ & & s_2^{-1} & & \\ & & & s_3^{-1} & \\ & & & & s_4^{-1} \end{bmatrix}}^{S^{-1}} \overbrace{\begin{bmatrix} \alpha_0 & 1 & & & \\ \beta_1^2 & \alpha_1 & 1 & & \\ & \beta_2^2 & \alpha_2 & 1 & \\ & & \beta_3^2 & \alpha_3 & 1 \\ & & & \beta_4^2 & \alpha_4 \end{bmatrix}}^{J} \overbrace{\begin{bmatrix} s_0 & & & & \\ & s_1 & & & \\ & & s_2 & & \\ & & & s_3 & \\ & & & & s_4 \end{bmatrix}}^{S}$$

$$= \underbrace{\begin{bmatrix} \alpha_0 & \frac{s_1}{s_0} & & & \\ \frac{s_0}{s_1}\beta_1^2 & \alpha_1 & \frac{s_2}{s_1} & & \\ & \frac{s_1}{s_2}\beta_2^2 & \alpha_2 & \frac{s_3}{s_2} & \\ & & \frac{s_2}{s_3}\beta_3^2 & \alpha_3 & \frac{s_4}{s_3} \\ & & & \frac{s_3}{s_4}\beta_4^2 & \alpha_4 \end{bmatrix}}_{\mathcal{J}}.$$

Verify that the matrix $\mathcal{J}$ on the right hand side has the same eigenvalues as J. We wish to choose the s_k's in such a way that $\mathcal{J}$ is symmetric. This is the case if, for each k,

$$\frac{s_{k-1}}{s_k}\beta_k^2 = \frac{s_k}{s_{k-1}}.$$

[3] Note that $v(x) \neq 0$ since its first entry is 1.

Since $\beta_k^2 = \frac{\|\phi_k\|_2^2}{\|\phi_{k-1}\|_2^2}$, a valid choice is:

$$s_k = \|\phi_k\|_2.$$

The resulting symmetric matrix is:

$$\mathcal{J} = \begin{bmatrix} \alpha_0 & \beta_1 & & & \\ \beta_1 & \alpha_1 & \beta_2 & & \\ & \beta_2 & \alpha_2 & \beta_3 & \\ & & \beta_3 & \alpha_3 & \beta_4 \\ & & & \beta_4 & \alpha_4 \end{bmatrix}.$$

This is indeed tridiagonal and symmetric, and has the same eigenvalues as J, so that its eigenvalues are the roots of ϕ_5. We can generalize.

Theorem 9.6. *The roots of ϕ_{n+1} are the eigenvalues of the matrix*

$$\mathcal{J}_{n+1} = \begin{bmatrix} \alpha_0 & \beta_1 & & & \\ \beta_1 & \alpha_1 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{bmatrix}. \tag{9.6}$$

The eigenvalues give us the roots. To get the weights, for now, we only know one way: solve the ill-conditioned Vandermonde system (9.1). Fortunately, there is a better way. It involves the eigenvectors of $\mathcal{J}_{n+1}$.

Remark 9.7. *As a side note, Cauchy's interlace theorem, Theorem 5.18, further tells us the roots of a system of orthogonal polynomials interlace since $\beta_k \neq 0$ for all k. This is not surprising, given the resemblance between the recurrences (5.19) and (8.6).*

9.4.2 Getting the weights, too: Golub–Welsch

The Gauss quadrature rule now looks like this:

$$\int_a^b f(x)w(x)\mathrm{d}x \approx \sum_{j=0}^{n} w_j f(x_j),$$

where $x_0 < \cdots < x_n$ are the eigenvalues of $\mathcal{J}_{n+1}$, and the weights can be chosen to ensure exact integration if $f \in \mathcal{P}_{2n+1}$.

This last point leads to the following observation. Define a[4] system of ortho*normal* polynomials:

$$\varphi_k(x) = \frac{1}{\|\phi_k\|_2}\phi_k(x), \qquad \text{for } k = 0, 1, 2, \ldots$$

Then, for any k, ℓ in $0, \ldots, n$,

$$\begin{aligned}\delta_{k\ell} &= \langle \varphi_k, \varphi_\ell \rangle \\ &= \int_a^b \varphi_k(x)\varphi_\ell(x)w(x)\mathrm{d}x \\ &\overset{!}{=} \sum_{j=0}^{n} w_j \varphi_k(x_j)\varphi_\ell(x_j).\end{aligned}$$

This last equality[5] follows from the fact that $\varphi_k(x)\varphi_\ell(x)$ is a polynomial of degree at most $2n$; thus, it is integrated *exactly* by the Gauss quadrature rule. Verify that these equations can be written in matrix form:

$$I = PWP^T,$$

with I the identity matrix of size $n+1$ and

$$P = \begin{bmatrix} \varphi_0(x_0) & \cdots & \varphi_0(x_n) \\ \vdots & & \vdots \\ \varphi_n(x_0) & \cdots & \varphi_n(x_n) \end{bmatrix}, \qquad W = \begin{bmatrix} w_0 & & \\ & \ddots & \\ & & w_n \end{bmatrix}.$$

The identity $I = PWP^T$ notably implies that both P and W are invertible. Hence,

$$W = P^{-1}(P^T)^{-1} = (P^TP)^{-1}.$$

Alternatively,

$$W^{-1} = P^TP.$$

In other words:

$$\frac{1}{w_j} = (P^TP)_{jj} = \sum_{k=0}^{n} P_{kj}^2 = \sum_{k=0}^{n} (\varphi_k(x_j))^2. \tag{9.7}$$

[4] Unique if we further require the leading coefficient to be positive, as is the case here.

[5] This equality also shows that $\varphi_0, \varphi_1, \ldots$ are not only orthonormal with respect to a continuous inner product, but also with respect to a discrete inner product (see also Remark 9.10). But this is a story for another time.

How do we compute the right hand side? This is where the eigenvectors come in.

Recall $Jv(x) = xv(x)$ from eq. (9.5). Apply S^{-1} on the left and insert SS^{-1} to find:

$$\underbrace{S^{-1}JS}_{\mathcal{J}}\underbrace{S^{-1}v(x)}_{u(x)} = x\underbrace{S^{-1}v(x)}_{u(x)}.$$

Thus, $u(x)$ is an eigenvector of $\mathcal{J}$ associated to the eigenvalue x, and

$$u(x) = S^{-1}v(x) = \begin{bmatrix} \frac{1}{\|\phi_0\|_2} & & & & \\ & \frac{1}{\|\phi_1\|_2} & & & \\ & & \frac{1}{\|\phi_2\|_2} & & \\ & & & \frac{1}{\|\phi_3\|_2} & \\ & & & & \frac{1}{\|\phi_4\|_2} \end{bmatrix} \begin{bmatrix} \phi_0(x) \\ \phi_1(x) \\ \phi_2(x) \\ \phi_3(x) \\ \phi_4(x) \end{bmatrix} = \begin{bmatrix} \varphi_0(x) \\ \varphi_1(x) \\ \varphi_2(x) \\ \varphi_3(x) \\ \varphi_4(x) \end{bmatrix}.$$

Corollary 9.8. *The roots of φ_{n+1} (equivalently, the roots of ϕ_{n+1}) are the eigenvalues of the symmetric, tridiagonal matrix $\mathcal{J}_{n+1}$. Let $x_0 < \cdots < x_n$ in (a, b) denote these roots. Any eigenvector $u^{(j)}$ associated to the eigenvalue x_j is of the form:*

$$u^{(j)} = c_j \begin{bmatrix} \varphi_0(x_j) \\ \varphi_1(x_j) \\ \vdots \\ \varphi_n(x_j) \end{bmatrix}, \tag{9.8}$$

for some constant c_j.

Let $u^{(j)}$ be an eigenvector of $\mathcal{J}_{n+1}$ associated to the eigenvalue x_j as in (9.8), with constant c_j chosen so that $u^{(j)}$ is *normalized*,[6] that is,

$$1 = \|u^{(j)}\|_2^2 = \sum_{k=0}^{n}(u_k^{(j)})^2 = c_j^2\sum_{k=0}^{n}(\varphi_k(x_j))^2.$$

Then, going back to (9.7), we find

$$w_j = c_j^2.$$

This brings us to the final question: how do we find c_j? Given an eigenvector, it is easy to normalize it, but that doesn't tell us what c_j is. The trick is

[6] This is with respect to the *vector* 2-norm now!

to observe that φ_0 is a *constant*. Specifically, it is the constant polynomial whose norm is 1. So, for some fixed μ, we have:

$$\varphi_0(x) = \mu, \qquad \forall x.$$

Then,

$$u_0^{(j)} = c_j \varphi_0(x_j) = c_j \mu.$$

Thus, to find c_j (which gives us w_j), we only need to find μ. To this end, note that,

$$\|\varphi_0\|_2 = 1 \iff \int_a^b \mu^2 w(x) \mathrm{d}x = 1,$$

so that

$$\mu^2 = \frac{1}{\int_a^b w(x)\mathrm{d}x}.$$

Thus,

$$w_j = c_j^2 = \frac{(u_0^{(j)})^2}{\mu^2} = (u_0^{(j)})^2 \underbrace{\int_a^b w(x)\mathrm{d}x}_{\text{compute once}}. \tag{9.9}$$

After these derivations, the practical implications are clear: this last equation is all you need to figure out how to implement the Golub–Welsch algorithm. This algorithm computes the nodes and the weights of a Gaussian quadrature: see Algorithm 9.1.

Question 9.9. *Implement the procedure above, known as the Golub–Welsch algorithm, to compute the nodes and weights of the Gauss–Legendre quadrature rule (Gauss with Legendre polynomials, that is, $w(x) = 1$ over $[-1, 1]$.)*

■

Remark 9.10. *The procedure above also proves the weights in a Gauss quadrature rule are positive. This is great numerically. Indeed, assume you are integrating $f(x)$ which is nonnegative over $[a, b]$. Then, the quadrature rule is a sum of nonnegative numbers: there is no risk of catastrophic cancellation due to round-off errors in computing the sum. This is not so for Newton–Cotes rules, which have negative weights already for $n \geq 8$: these may lead to intermediate computations of differences of large numbers.*

Algorithm 9.1 Golub–Welsch for Gauss quadrature nodes and weights

1: Given: $\alpha_0, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$: the three-term recurrence coefficients for the system of polynomials orthogonal with respect to weight $w(x)$ over $[a, b]$; see Remark 8.18 for example. Furthermore, let $t = \int_a^b w(x)\mathrm{d}x$.
2: Construct the symmetric, tridiagonal matrix $\mathcal{J}_{n+1}$ as in (9.6).
3: For the matrix $\mathcal{J}_{n+1}$, compute the eigenvalues $x_0 < \cdots < x_n$ in (a, b) and the associated unit-norm eigenvectors $u^{(0)}, \ldots, u^{(n)}$ in $\mathbb{R}^{n+1}$.
4: Following (9.9), for $j = 0, \ldots, n$, let $w_j = (u_0^{(j)})^2 \cdot t$, where $u_0^{(j)}$ is the first entry of eigenvector $u^{(j)}$.

9.4.3 Examples

Figures 9.1–9.4 compare different quadrature rules, obtained in different ways, on four different integrals.

- Newton–Cotes (Vandermonde) uses $n + 1$ equispaced points on the interval, and obtains the weights by solving the Vandermonde system (ill conditioned).
- Chebyshev nodes (Vandermonde) does the same but with $n+1$ Chebyshev nodes in the interval.
- Legendre nodes (Vandermonde) does the same but using the $n + 1$ roots of the $(n + 1)$st Legendre polynomial. Mathematically, the later is the Gauss quadrature with $n + 1$ points for unit weight $w(x) = 1$, but because solving the Vandermonde system is unstable, it eventually fails.
- Legendre nodes (Golub–Welsch) is the same rule as the previous one, but it computes the weights with Golub–Welsch, which is far more trustworthy numerically.
- The composite trapezium rule uses n trapezia to approximate the integral.

At point n, all rules use exactly $n + 1$ evaluations of the integrand. They require varying amounts of work to compute the nodes and weights, but notice that these can be precomputed once and for all.

9.4.4 Error bounds

We did not get a chance to discuss error bounds for Gauss quadrature rules in class. See Theorems 10.1 and 10.2 in [SM03] for results in this direction

(for your information.) Interestingly, the first one is related to approximation errors when interpolating à la Hermite—this is actually another way to construct Gauss quadratures, and is the way preferred in [SM03]. In the present lecture notes, we followed an approach which emphasizes the role of orthogonal polynomials instead, because (a) it is practical, (b) it is beautiful, and (c) this is the way to go to generalize Gauss quadratures to other domains besides the real line. The following is a side-note similar to the reasoning in Theorem 10.2 of [SM03].

Say we want to compute $\int_a^b f(x)\mathrm{d}x$ with f continuous (the story generalizes to include weights $w(x)$, but we here consider $w(x) = 1$ over $[a, b]$ for simplicity). Consider $p_{2n+1} \in \mathcal{P}_{2n+1}$: the minimax approximation of f over $[a, b]$ of degree at most $2n + 1$. We do not need to compute this polynomial: we only need to know that it exists, as we proved two chapters ago.

We can split f as follows:

$$f = p_{2n+1} + e_{2n+1},$$

where e_{2n+1} is some function (not necessarily a polynomial), and $\|e_{2n+1}\|_\infty$ becomes arbitrarily close to 0 as n goes to infinity, because polynomials are dense in $C[a, b]$ (Weierstrass' theorem).

Now say $x_0, \ldots, x_n$ and $w_0, \ldots, w_n$ form the Gauss quadrature rule of degree of precision $2n + 1$. Apply this rule to f:

$$\begin{aligned}\sum_{k=0}^{n} w_k f(x_k) &= \sum_{k=0}^{n} w_k p_{2n+1}(x_k) + \sum_{k=0}^{n} w_k e_{2n+1}(x_k) \\ &= \int_a^b p_{2n+1}(x)\mathrm{d}x + \sum_{k=0}^{n} w_k e_{2n+1}(x_k),\end{aligned}$$

where we used the fact that the quadrature rule is exact when applied to p_{2n+1}. On the other hand:

$$\int_a^b f(x)\mathrm{d}x = \int_a^b p_{2n+1}(x)\mathrm{d}x + \int_a^b e_{2n+1}(x)\mathrm{d}x.$$

Thus, our integration error is given by:

$$\begin{aligned}\left|\int_a^b f(x)\mathrm{d}x - \sum_{k=0}^{n} w_k f(x_k)\right| &= \left|\int_a^b e_{2n+1}(x)\mathrm{d}x - \sum_{k=0}^{n} w_k e_{2n+1}(x_k)\right| \\ &\leq \left|\int_a^b e_{2n+1}(x)\mathrm{d}x\right| + \left|\sum_{k=0}^{n} w_k e_{2n+1}(x_k)\right| \\ &\leq \|e_{2n+1}\|_\infty \int_a^b 1\mathrm{d}x + \|e_{2n+1}\|_\infty \sum_{k=0}^{n} |w_k| \\ &= 2(b-a)\|f - p_{2n+1}\|_\infty.\end{aligned}$$

The last step relies on two facts:

1. $w_0, \ldots, w_n \geq 0$ (see Remark 9.10), and
2. $w_0 + \ldots + w_n = b - a$ since the rule is exact for $f(x) = 1$.

Here is the take-away: the integration error of the Gauss quadrature rule is determined by the minimax error of approximating f with a polynomial of degree $2n+1$ (even though we are using only $n+1$ evaluations of f!). The error goes to zero with $n \to \infty$. If f lends itself to good minimax approximations of low degree, then the results should be quite good already for finite n (and remember: we do not need to compute the minimax approximation: we only use the fact that it exists). Furthermore, we can get an upper-bound on the bound by plugging in any other polynomial of degree at most $2n+1$ which is a good ∞-norm approximation of f. For example, we know that the polynomial of degree $2n+1$ which interpolates f at the $2n+2$ Chebyshev nodes on (a, b) reaches a pretty good ∞-norm error as long as f is many times continuously differentiable and those derivatives are not catastrophically crazy. We do at least as well as if we were using that polynomial, even though finding that polynomial would have required $2n+2$ evaluations of f!

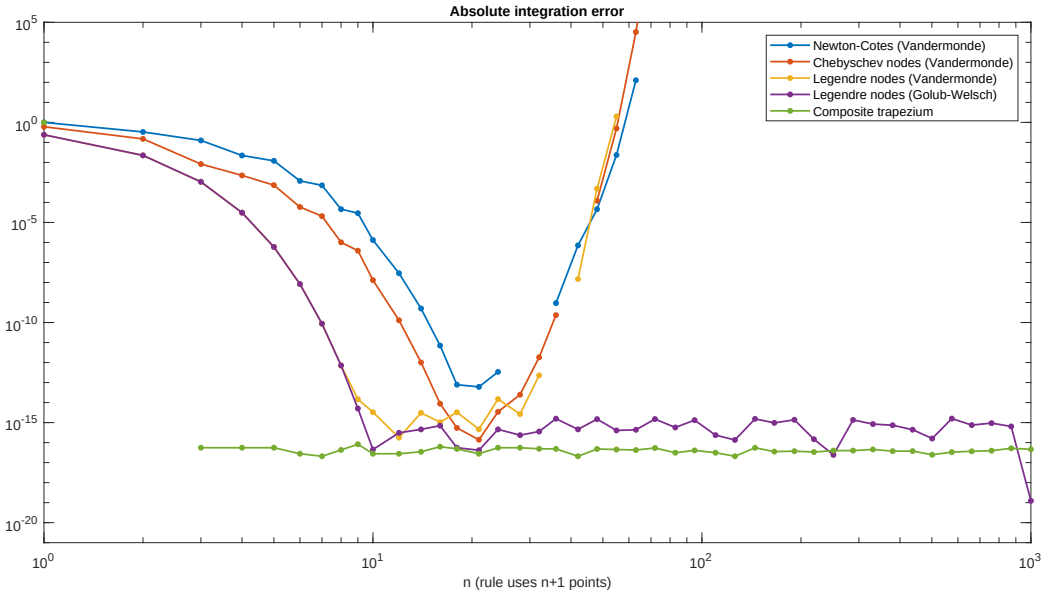


Figure 9.1: Computation of $\int_0^1 \cos(2\pi x)\mathrm{d}x = 0$ with various quadratures.

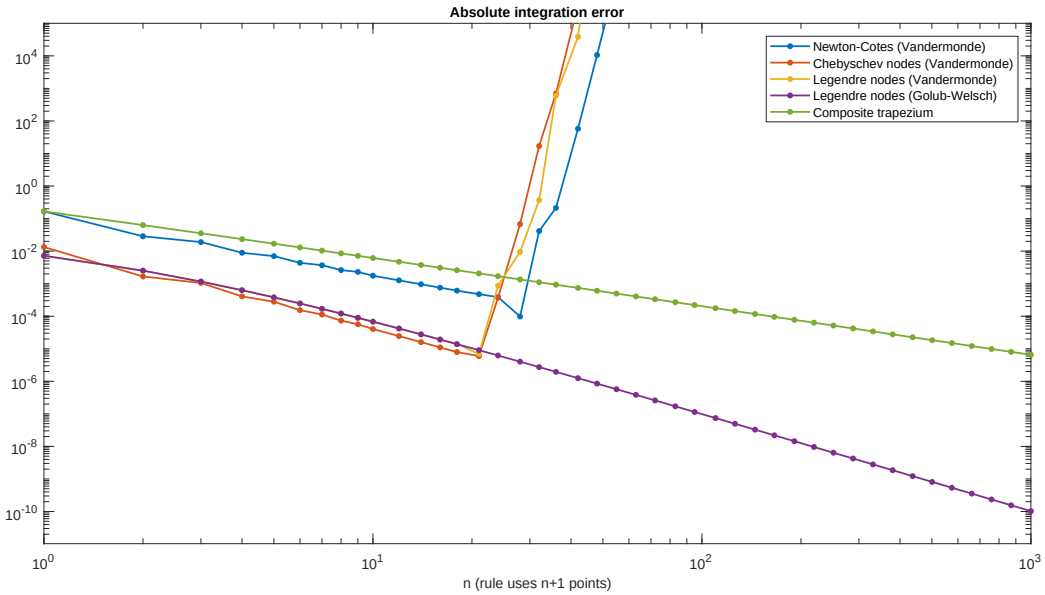


Figure 9.2: Computation of $\int_0^1 \sqrt{x}\mathrm{d}x = \frac{2}{3}$ with various quadratures.

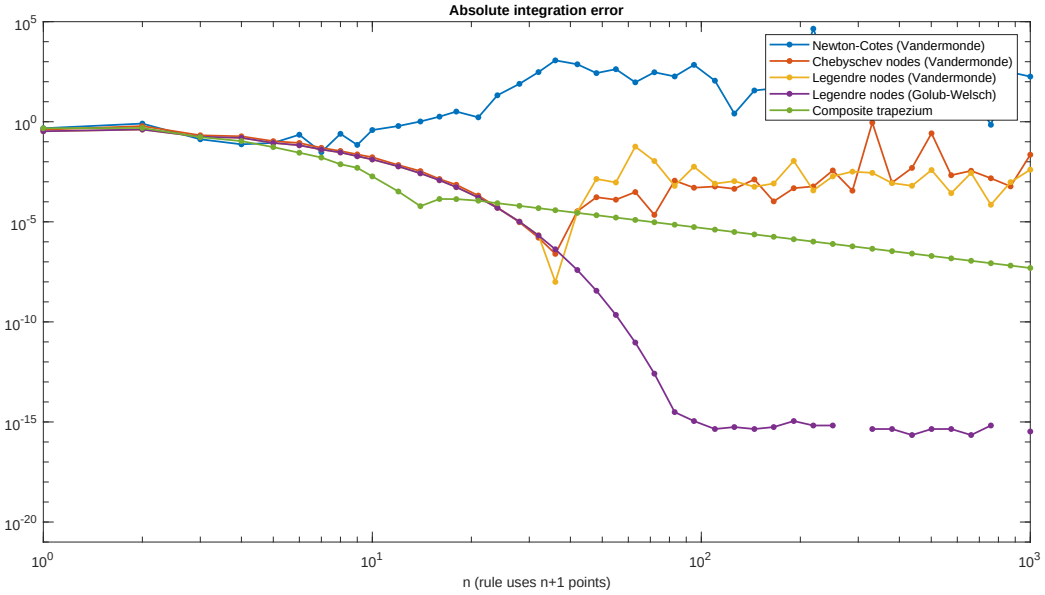


Figure 9.3: Computation of $\int_{-1}^{1} \frac{1}{1+25x^2}\mathrm{d}x = \frac{2}{5}\arctan(5)$ with various quadratures.

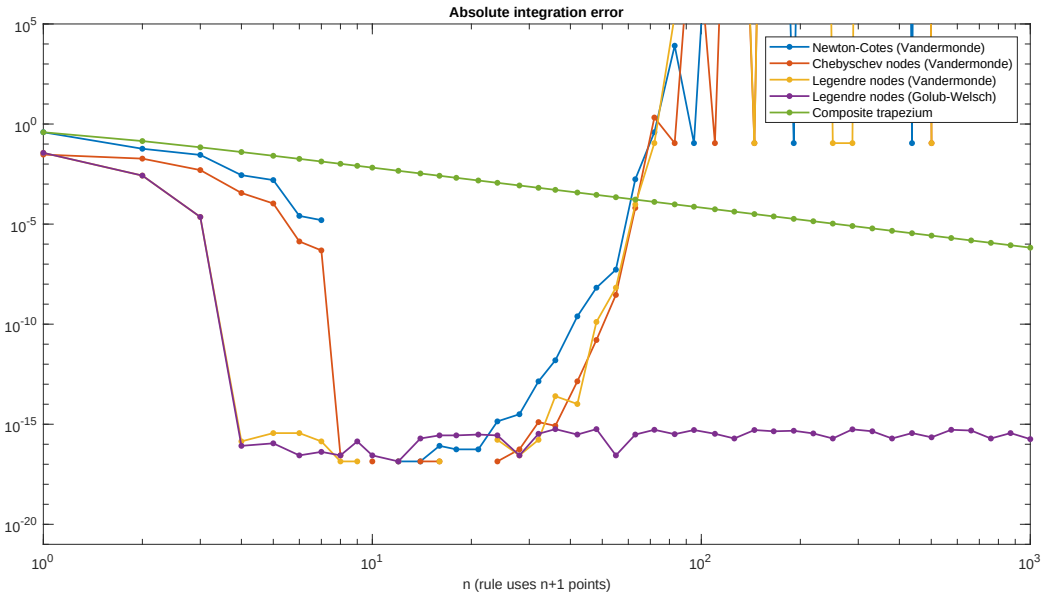


Figure 9.4: Computation of $\int_0^1 x^8\mathrm{d}x = \frac{1}{9}$ with various quadratures.

Chapter 10

Unconstrained optimization

One of the central problems in optimization is the following.

Problem 10.1. *Given a continuous function $f\colon \mathbb{R}^n \to \mathbb{R}$, compute*[1]

$$\min_{x\in\mathbb{R}^n} f(x), \tag{10.1}$$

that is, compute the minimal value that f can take for any choice of x.

In general, this value may not exist and may not be attainable, in which case the problem is less interesting. We make the following blanket assumption throughout the chapter to avoid this issue.

Assumption 10.2. *The function f is twice continuously differentiable*[2] *and attains the minimal value f^*.*

When the minimum exists and is attainable, one is often also interested in determining an $x^* \in \mathbb{R}^n$ for which this minimal value is attained. Such an x^* is called an *optimum*.[3] The *set* of all optima is denoted

$$\arg\min_{x\in\mathbb{R}^n} f(x). \tag{10.2}$$

In general, this set can contain any number of elements. Because the variable x is free to take up any value, we say the problem is *unconstrained*. It is important to make a distinction between *globally* optimal points and points that appear optimal only *locally* (that is, when compared only to their immediate surroundings.)

[1] It is traditional to talk about minimization. To maximize, consider $-f(x)$ instead.

[2] Many statements hold without demanding this much smoothness. Given the limited time at our disposal, we will keep technicality low to focus on ideas.

[3] Or also: an optimizer, a minimum, a minimizer.

Definition 10.3 (optimum and local optimum)**.** *A point $x \in \mathbb{R}^n$ is a* local optimum *of f if there exists a neighborhood U of x such that $f(x) \leq f(y)$ for all $y \in U$. A local optimum is a (global) optimum if $f(x) \leq f(y)$ for all $y \in \mathbb{R}^n$.*

The definition of optimum is exactly what we want, but it is not revealing in terms of computations. To work our way toward practical algorithms, we now determine *necessary optimality conditions*, that is: properties that optima must satisfy.

Lemma 10.4 (Necessary optimality conditions)**.** *If $x^* \in \mathbb{R}^n$ is an optimum of f, then $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*) \succeq 0$.*

Proof. For contradiction, assume $\nabla f(x^*) \neq 0$ and consider $x = x^* - \alpha \nabla f(x^*)$ for some $\alpha > 0$. By a Taylor expansion at x^*, we find that

$$f(x) = f(x^*) + \nabla f(x^*)^T (x - x^*) + o(\|x - x^*\|_2) \tag{10.3}$$
$$= f(x^*) - \alpha \|\nabla f(x^*)\|_2^2 + o(\alpha). \tag{10.4}$$

By definition, $o(\alpha)$ is a term such that $\lim_{\alpha \to 0} \frac{o(\alpha)}{\alpha} = 0$. Hence, there exists $\bar{\alpha} > 0$ such that for all $\alpha \in (0, \bar{\alpha})$ the term $-\alpha \|\nabla f(x^*)\|_2^2 + o(\alpha)$ is negative. This shows that $f(x) < f(x^*)$ which is a contradiction.

Similarly, assume now for contradiction that $\nabla^2 f(x^*) \not\succeq 0$. Thus, there exists $u \in \mathbb{R}^n$ of unit norm such that $u^T \nabla^2 f(x^*) u = -\delta < 0$. Using the fact that $\nabla f(x^*) = 0$ as we just established and a Taylor expansion, we find for $x = x^* + \alpha u$ that

$$f(x) = f(x^*) + (x - x^*)^T \nabla^2 f(x^*)(x - x^*) + o(\|x - x^*\|_2^2) \tag{10.5}$$
$$= f(x^*) - \alpha^2 \delta + o(\alpha^2). \tag{10.6}$$

Again, there is an interval $(0, \bar{\alpha})$ of values of α which show x outperforms x^*, leading to a contradiction. □

Notice that local optima also are second-order critical points. It is useful to give a name to all points which satisfy the necessary conditions.

Definition 10.5 (Critical points)**.** *A point $x \in \mathbb{R}^n$ is a* (first-order) critical point *if $\nabla f(x) = 0$. If furthermore $\nabla^2 f(x) \succeq 0$, then x is a* second-order critical point*.*

Critical points which are neither local minima nor local maxima are called *saddle points*.

As will become clear, in general, we can only hope to compute first- or second-order critical points. Since these can be arbitrarily far from optimal,

this is a major obstacle to our endeavor. Fortunately, for certain classes of functions, it is the case that all critical points are optima, so that the necessary conditions of Lemma 10.4 are also sufficient.

Definition 10.6 (convex function)**.** *A function $g\colon \mathbb{R}^n \to \mathbb{R}$ is convex if, for all $x, y \in \mathbb{R}^n$ and for all $\alpha \in [0,1]$,*

$$g(\alpha x + (1-\alpha)y) \leq \alpha g(x) + (1-\alpha)g(y), \tag{10.7}$$

that is, if the line segment between any two points on the graph of the function lies above or on the graph.

Lemma 10.7. *Owing to the differentiability properties of f, these are equivalent:*

(a) f is convex;

(b) $f(y) \geq f(x) + (y-x)^T \nabla f(x)$ for all $x, y \in \mathbb{R}^n$;

(c) $\nabla^2 f(x) \succeq 0$ for all $x \in \mathbb{R}^n$.

Proof. We first show (a) and (b) are equivalent.

If f is convex, then for any $\alpha \in [0,1]$ we have

$$f(x + \alpha(y-x)) \leq \alpha f(y) + (1-\alpha) f(x),$$

or, equivalently,

$$\frac{f(x + \alpha(y-x)) - f(x)}{\alpha} \leq f(y) - f(x).$$

Furthermore, by definition of the gradient,

$$\lim_{\alpha \downarrow 0} \frac{f(x + \alpha(y-x)) - f(x)}{\alpha} = (y-x)^T \nabla f(x).$$

Thus, $(y-x)^T \nabla f(x) \leq f(y) - f(x)$, as desired.

The other way around, assume now that (b) holds. For any $x, y \in \mathbb{R}^n$ and any $\alpha \in [0,1]$, consider $z = \alpha x + (1-\alpha)y$. Using (b) twice we get

$$\begin{aligned} f(x) &\geq f(z) + (x-z)^T \nabla f(z), \\ f(y) &\geq f(z) + (y-z)^T \nabla f(z). \end{aligned}$$

Multiply the first by α and the second by $1-\alpha$ and add:

$$\begin{aligned} \alpha f(x) + (1-\alpha) f(y) &\geq f(z) + (\alpha x + (1-\alpha)y - z)^T \nabla f(z) \\ &= f(z) \\ &= f(\alpha x + (1-\alpha)y), \end{aligned}$$

as desired.

We now show that (b) and (c) are equivalent.

If (c) holds, then simply consider a Taylor expansion of f: for any $x, y \in \mathbb{R}^n$, there exists $\alpha \in [0,1]$ such that

$$f(y) = f(x) + (y-x)^T \nabla f(x) + \frac{1}{2}(y-x)^T \nabla^2 f(x + \alpha(y-x))(y-x). \tag{10.8}$$

Since the Hessian is assumed everywhere positive semidefinite, this yields

$$f(y) \geq f(x) + (y-x)^T \nabla f(x)$$

for all x, y, showing (b) holds.

The other way around, if (b) holds, then (c) must hold. Assume otherwise for contradiction. Then, there exists $x \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$ such that $u^T \nabla^2 f(x) u < 0$. By continuity of the Hessian, we can take u small enough so that $u^T \nabla^2 f(x + \alpha u) u < 0$ for all $\alpha \in [0,1]$ as well. Plugging this in the Taylor expansion (10.8) with $y = x + u$, we find

$$\begin{aligned} f(y) &= f(x) + (y-x)^T \nabla f(x) + \frac{1}{2} u^T \nabla^2 f(x + \alpha u) u \\ &< f(x) + (y-x)^T \nabla f(x), \end{aligned}$$

which indeed contradicts (b). □

Question 10.8. *Show that linear functions, vector norms and weighted sums of convex functions with positive weights are convex. Show that if $g \colon \mathbb{R}^n \to \mathbb{R}$ is convex, then g composed with a linear function is convex. Show that if $g_1, \ldots, g_n \colon \mathbb{R}^n \to \mathbb{R}$ are convex, then $g(x) = \sup_{i \in \{1,\ldots,n\}} g_i(x)$ is convex.*

Lemma 10.9. *If f is convex, then critical points and minima coincide.*

Proof. If $\nabla f(x) = 0$, use part (b) of Lemma 10.7 to conclude that x is optimal. The other way around, if x is optimal, it is a critical point by Lemma 10.4, so that $\nabla f(x) = 0$. □

10.1 A first algorithm: gradient descent

Gradient descent (or steepest descent) is probably the most famous and most versatile optimization algorithm—take a look at Algorithm 10.1. The algorithm is also called steepest descent because, locally, following the negative gradient induces the steepest decrease in the cost function f (up to first order approximation).

Algorithm 10.1 Gradient descent

Input: $x_0 \in \mathbb{R}^n$
for $k = 0, 1, 2 \ldots$ **do**
    Pick a step-size η_k
    $x_{k+1} = x_k - \eta_k \nabla f(x_k)$
end for

In general, we can only guarantee convergence to critical points,[4] but it is intuitively expected that the algorithm converges to a local minimum unless the initial point x_0 is chosen maliciously—an intuition which can be formalized.

Besides an initial guess $x_0 \in \mathbb{R}^n$, gradient descent requires a strategy to pick the step-size η_k. This amounts to considering the line-search problem

$$\min_{\eta \in \mathbb{R}} \phi(\eta) = f(x_k - \eta \nabla f(x_k)), \tag{10.9}$$

which restricts f to a one-dimensional problem along the line generated by x_k and $\nabla f(x_k)$. At first sight, it might seem like a good idea to pick $\eta \in \arg\min_\eta \phi(\eta)$, but this is impractical for all but the simplest settings.

Fortunately, there is no need to solve the line-search problem exactly to ensure convergence. It is enough to ensure *sufficient decrease*.

Definition 10.10. *We say the step-sizes in gradient descent yield sufficient decrease if there exists $c > 0$ such that, for all k,*

$$f(x_k) - f(x_{k+1}) \geq c \|\nabla f(x_k)\|_2^2. \tag{10.10}$$

Lemma 10.11. *If the step-sizes yield sufficient decrease, then gradient descent produces an iterate x_k such that $\|\nabla f(x_k)\|_2 \leq \varepsilon$ with $k \leq \lceil \frac{f(x_0) - f^*}{c} \frac{1}{\varepsilon^2} \rceil$ and $\|\nabla f(x_k)\|_2 \to 0$. There is no condition on x_0.*

Proof. Assume that $x_0, \ldots, x_{K-1}$ all have gradient larger than ε. Then, using both the fact that f is lower bounded and the sufficient decrease property, a classic telescoping sum argument gives

$$\begin{aligned} f(x_0) - f^* &\geq f(x_0) - f(x_K) \\ &= \sum_{\ell=0}^{K-1} f(x_\ell) - f(x_{\ell+1}) \geq \sum_{\ell=0}^{K-1} c \|\nabla f(x_\ell)\|_2^2 \geq cK\varepsilon^2. \end{aligned}$$

[4] Notice the plural: we won't show convergence to a unique critical point, even though this is what typically happens in practice. We only show that all accumulation points are critical points.

Hence, $K \leq (f(x_0) - f^*)/c\varepsilon^2$, so that if more iterations are computed, it must be that the gradient dropped below ε at least once. Furthermore, since the sum of squared gradient norms is upper bounded, the gradient norm must converge to 0. □

Remark 10.12. *Let us stress this: there are* no *assumptions on x_0. On the other hand, the theorem only guarantees that all accumulation points of the sequence of iterates are critical points: it does not guarantee that these are global optima. Importantly, if f is convex, then critical points and global optima coincide, which shows all accumulation points are global optima regardless of initialization: this is powerful!*

Sufficient decrease can be achieved easily if we assume f has a Lipschitz continuous gradient with known constant L.

Lemma 10.13. *If $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$ for all x, y, then the constant step size $\eta_k = \frac{1}{L}$ yields sufficient decrease with $c = \frac{1}{2L}$.*

Proof. First, we show the Lipschitz condition implies the following statement: for all x, y,

$$\left|f(y) - f(x) - (y - x)^T \nabla f(x)\right| \leq \frac{L}{2}\|y - x\|_2^2. \tag{10.11}$$

Indeed, by the fundamental theorem of calculus,

$$\begin{aligned} f(y) - f(x) &= \int_0^1 (y - x)^T \nabla f(x + s(y - x)) \mathrm{d}s \\ &= (y - x)^T \nabla f(x) + \int_0^1 (y - x)^T \big[\nabla f(x + s(y - x)) - \nabla f(x)\big] \mathrm{d}s. \end{aligned}$$

The error term is easily bounded:

$$\begin{aligned} &\left| \int_0^1 (y - x)^T \big[\nabla f(x + s(y - x)) - \nabla f(x)\big] \mathrm{d}s \right| \\ &\leq \|y - x\|_2 \int_0^1 \|\nabla f(x + s(y - x)) - \nabla f(x)\|_2 \, \mathrm{d}s \\ &\leq \|y - x\|_2 \int_0^1 L\|s(y - x)\|_2 \mathrm{d}s \\ &= \frac{L}{2}\|y - x\|_2^2, \end{aligned}$$

which confirms (10.11). Let η be our tentative step-size, so that $x_{k+1} = x_k - \eta \nabla f(x_k)$. Then, equation (10.11) can be used to bound the improvement

one can expect: set $x = x_k$ and $y = x_{k+1}$, then, removing the absolute value on the left-hand side,

$$f(x_{k+1}) - f(x_k) + \eta \|\nabla f(x_k)\|_2^2 \leq \frac{L}{2}\eta^2 \|\nabla f(x_k)\|_2^2,$$

or, equivalently,

$$f(x_k) - f(x_{k+1}) \geq \eta \left(1 - \frac{L}{2}\eta\right) \|\nabla f(x_k)\|_2^2. \tag{10.12}$$

The right-hand side is largest if $\eta = \frac{1}{L}$, at which point the improvement at every step is at least $\frac{1}{2L}\|\nabla f(x_k)\|_2^2$ as announced. □

Notice that we only use the Lipschitz property along the piecewise linear curve that joins the iterates $x_0, x_1, x_2 \ldots$. This can help when analyzing functions f whose gradients are not globally Lipschitz continuous.

Algorithm 10.2 Backtracking Armijo line-search

Given: $x_k \in \mathbb{R}^n$, $c_1 \in (0,1)$, $\tau \in (0,1)$, $\bar{\eta} > 0$
Init: $\eta \leftarrow \bar{\eta}$
while $f(x_k) - f(x_k - \eta \nabla f(x_k)) < c_1 \eta \|\nabla f(x_k)\|_2^2$ **do**
    $\eta \leftarrow \tau \cdot \eta$
end while
return $\eta_k = \eta$.

In practice, one rarely knows the Lipschitz constant L, and it is standard to use a line-search algorithm, such as Algorithm 10.2 for example. The reasoning is as follows: based on (10.12), it is clear that any choice of η_k in the open interval $(0, 2/L)$ guarantees decrease in the cost function. To further ensure sufficient decrease, it is merely necessary to find a step-size in that interval which remains bounded away from both ends of the interval. Algorithm 10.2 achieves this without knowledge of L, in a logarithmic number of steps. (Again: the constant L appears in the bound of the lemma, but needs not be known to run the algorithm.) Another advantage is adaptivity: in regions where f is less flat, the method will dare make larger steps. Of course, each iteration of Algorithm 10.2 involves one call to f, which may be costly. In practice, a lot of engineering goes into fine-tuning the line-search algorithm. For example, a nice touch is to "remember" the previous step-size and to use it to better pick $\bar{\eta}$ at the next iterate.

Lemma 10.14. *Assume f has L-Lipschitz continuous gradient. The backtracking line-search Algorithm 10.2 returns a step-size η which satisfies*

$$\eta \geq \min\left(\bar{\eta}, \frac{2(1-c_1)\tau}{L}\right)$$

in at most

$$\max\left(1, 2 + \log_{\tau^{-1}}\left(\frac{L\bar{\eta}}{2(1-c_1)}\right)\right)$$

calls to the cost function f. In particular, this means the sufficient decrease condition (Def. 10.10) is met with

$$c \geq \min\left(c_1\bar{\eta}, \frac{2c_1(1-c_1)\tau}{L}\right),$$

implying convergence of Algorithm 10.1. (Notice that the right-hand side is always smaller than $1/2L$.)

Proof. When the line-search algorithm tries the step-size η, the Lipschitz continuous gradient assumption, via (10.11), guarantees that

$$f(x_k) - f(x_k - \eta\nabla f(x_k)) \geq \eta\left(1 - \frac{L}{2}\eta\right)\|\nabla f(x_k)\|_2^2.$$

(This is the same as (10.12).) If the algorithm does not stop, it must be that

$$\eta\left(1 - \frac{L}{2}\eta\right)\|\nabla f(x_k)\|_2^2 < c_1\eta\|\nabla f(x_k)\|_2^2,$$

or, equivalently, that $1 - \frac{L}{2}\eta < c_1$, so that

$$\eta > \frac{2(1-c_1)}{L}.$$

As soon as η drops below this bound, we can be sure that the line-search will return. This happens either at the very first guess, when $\eta = \bar{\eta}$, or after a step-size reduction by a factor τ, which cannot have reduced the step-size below τ times the right-hand side, so that, when the algorithm returns, η satisfies:

$$\eta \geq \min\left(\bar{\eta}, \frac{2(1-c_1)\tau}{L}\right). \tag{10.13}$$

In other terms: the returned step-size is at least as large as a certain constant in $(0, 2/L)$.

In the worst case, how many times might we need to call f before the line-search returns? After $\ell + 1$ calls to the function f, the backtracking line-search tries the step-size $\bar{\eta}\tau^\ell$. Either the line-search returns $\bar{\eta}$ after the first call to f, or it returns

$$\bar{\eta}\tau^\ell \geq \frac{2(1-c_1)}{L}\tau.$$

Hence, minding the fact that $\tau \in (0,1)$ so that $\log(\tau) < 0$, we find successively

$$(\ell - 1)\log(\tau) \geq \log\left(\frac{2(1-c_1)}{L\bar{\eta}}\right)$$
$$\ell + 1 \leq 2 + \frac{\log\left(\frac{2(1-c_1)}{L\bar{\eta}}\right)}{\log(\tau)}.$$

This concludes the proof, since $\ell + 1$ is here the number of calls to f. □

The main advantage of Lemma 10.11 is that it makes no assumptions about the initial iterate x_0 (compare this with our work in Chapters 1 and 4 of [SM03]). Yet, the guaranteed convergence rate is very slow: it is sublinear. This is because, in the worst case, f may present large, almost flat regions where progress is slow. Luckily, the convergence rate becomes linear if the iterates get sufficiently close to a (strict) local optimum. Here is a statement to that effect, with somewhat overly restrictive assumptions.

Lemma 10.15. *Under the Lipschitz assumptions on the gradient of f, if x^* is a local optimum where $0 \prec \nabla^2 f(x^*) \prec LI_n$, there exists a neighborhood U of x^* such that, if the sequence $x_0, x_1, x_2 \ldots$ generated by gradient descent with constant step-size $\eta_k = 1/L$ ever enters the neighborhood U, then the sequence converges at least linearly to x^*.*

Proof. Proof sketch: it is sufficient to observe that gradient descent in this setting is simultaneous iteration through relaxation:

$$x_{k+1} = g(x_k) = x_k - \frac{1}{L}\nabla f(x_k). \tag{10.14}$$

The Jacobian of g at the fixed-point x^* is $J_g(x^*) = I_n - \frac{1}{L}\nabla^2 f(x^*)$. Under the assumptions, the eigenvalues of $\nabla^2 f(x^*)$ all lie in $(0, L)$, so that $\|J_g(x^*)\|_2 < 1$. Thus, by continuity, g is a contraction map in a neighborhood of x^*. From there, one can deduce linear convergence of x_k to to x^* (after U has been entered.) □

Notice that x^* being a local optimum readily implies $\nabla^2 f(x^*) \succeq 0$ and the Lipschitz condition readily implies $\nabla^2 f(x^*) \preceq LI_n$. Linear convergence can also be established when the line-search algorithm is used.

> **The main take-away (informal):** if f is sufficiently smooth, gradient descent with appropriate step-sizes converges to critical points without conditions on the initial iterate x_0; while the convergence may be sublinear and to a saddle point, it is normally expected that the convergence rate will eventually be linear and to a local optimum. If f is convex, we get convergence to a global minimum.

10.2 More algorithms

Continuous optimization is a whole field of research, extending far beyond gradient descent. One important algorithm you actually already know: Newton's method. Indeed, for convex f at least, minimizing f is equivalent to finding x such that $\nabla f(x) = 0$: this is a system of (usually) nonlinear equations. Applying Newton's method to find roots of $\nabla f(x)$ leads to this iteration:

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k),$$

assuming the inverse exists. This step is computed by solving a linear system where the matrix is the Hessian of f at x_k. Importantly, one does *not* construct the Hessian matrix to do this. That would be very expensive in most applications. Instead, one resorts to matrix-free solvers, which only require the ability to compute products of the form $\nabla^2 f(x_k)u$ for vectors u.

Another interpretation of Newton's method for optimization is the following: at x_k, approximate f with a second-order Taylor expansion:

$$f(x) \approx f(x_k) + (x - x_k)^T \nabla f(x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k).$$

If f is strictly convex,[5] then this quadratic approximation of f is itself strictly convex. Find x which minimizes the quadratic: this coincides with x_{k+1}! In other words: an iteration of Newton's method for optimization consists in moving to the critical point of the quadratic approximation of f around the current iterate.

[5] Strictly convex means the Hessian is positive definite, rather than only positive semidefinite.

What if f is not strictly convex? Then, the quadratic may not be convex, so that its critical points is not necessarily a minimizer: it could be a maximizer, or a saddle point. If such is the case, then moving to that point is ill advised. A better strategy consists in recognizing that the Taylor expansion can only be trusted in a small neighborhood around x_k. Thus, the quadratic should be minimized in that trusted region only (instead of blindly jumping to the critical point.) This is the starting point of the so-called *trust region method*, which is an excellent algorithm widely used in practice.

What if we do not have access to the Hessian? One possibility is to approximate the Hessian using finite differences of the gradient. Alternatively, one can resort to the popular BFGS algorithm, which only requires access to the gradient and works great in practice as well.

Chapter 11

What now?

Let's take a quick look in the mirror. We discussed numerical algorithms to solve the following problems:

1. Solve $Ax = b$, for various kinds of A's, also in a least-squares sense. This lead us to consider LU and QR factorizations.

2. Solve $f(x) = 0$ for $f\colon \mathbb{R}^n \to \mathbb{R}$, with $n = 1$ or $n > 1$.

3. Compute eigenvalues and eigenvectors of a matrix A, $Ax = \lambda x$.

4. Interpolate or approximate a function f with polynomials, which lead us to the mesmerizing world of orthogonal polynomials: $f \approx p_n$.

5. Approximate derivatives and integrals of functions, $f'(x)$, $\int_a^b f(x)\mathrm{d}x$.

6. Compute the minimum of a function f, $\min_x f(x)$.

For most problems, we also assessed which aspects of it can make it harder or easier, through analysis. For example, $Ax = b$ is more difficult to solve if A is poorly conditioned, and f is more difficult to approximate with polynomials if its high-order derivatives go wild. We also acknowledged the effects of inexact arithmetic.

Hopefully, you were convinced that mathematical proofs and numerical experimentation inform each other. Both are necessary to gain confidence in the algorithms we develop, and eventually use in settings where failure has consequences.

What now? The problems we studied are fundamental, in that they appear throughout the sciences and engineering. I am confident that you will encounter these problems in your own work, in various ways. With even more certainty, you will encounter problems we did not address at all. Some

of these appear in chapters of [SM03] we did not open. By now, you are well equipped to study these problems and algorithms on your own.

1. Piecewise polynomial approximation (splines): instead of interpolating f with a polynomial of high degree, divide $[a, b]$ into smaller intervals, and approximate f with a low-degree polynomial on each interval separately. Only, do it in a way that the polynomials "connect": the piecewise polynomial function should be continuous, and continuously differentiable a number of times. These requirements lead to a banded linear system: you know how to solve this efficiently.

2. Initial value problems for ODEs: in such problems, the unknown is a function f, which has to satisfy a differential equation. If this equation is linear and homogeneous, then the solutions are obtained explicitly in terms of the eigenvectors and eigenvalues of the corresponding matrix. If the system is linear but inhomogeneous, then explicit solutions still exist if the inhomogeneous term is polynomial. If it is not polynomial, you know how to approximate it accurately with a polynomial. If the ODE is nonlinear, then stepping methods can be used: the gradient descent algorithm is actually an example of a stepping method (explicit Euler) applied to a specific ODE, so the concept is not too foreign. Some fo these stepping methods (implicit Euler) require solving a nonlinear equation at each step. In electrical engineering, ODEs dictate the behavior of electrical circuits with impressive accuracy. To simulate a circuit before sending it to production (a costly stage), state of the art industrial software solves these ODEs as efficiently and accurately as they can. ODEs are also used to model chemical reactions in industrial plants, ecosystems in predator-prey models, and so many other situations. The natural question that ensues is: can we control these systems? The answer is yes to a large extent, and leads to control theory, which ultimately relies on our capacity to solve ODEs numerically.

3. Boundary value problems for ODEs: in one example of this, a differential equation dictates the behavior of a system (for example, $F = \frac{\mathrm{d}}{\mathrm{d}t}(mv)$ dictates the ballistics of a space rocket), and the goal is to determine the initial conditions one should pick (fuel loaded in the tank, propeller settings, etc.) so that the system will attain a specified state at a specified time (for example, attain a geostationary orbit and stay there after a certain amount of time.) In this scenario, there exists a function f which maps the initial conditions (the variables) to the final conditions. The goal is to pick the variables such that the final

conditions are as prescribed; thus: we are looking for the roots of this function. It is nonlinear, and to evaluate it we must solve the initial value problem (for example using a stepping method): it is thus crucial to use a nonlinear equation solver which requires as few calls to the function as possible.

4. The finite element method (FEM) for ODEs and PDEs: this method is used extensively in materials sciences, mechanical engineering, geosciences, climate modeling and many more. For example, a differential equation dictates the various mechanical constraints on the wings of an airplane in flight, as a function of shape, wind speed and materials used. Solving this equation informs us about which specific points of the wing are at risk of breaking first, that is: which points should be engineered with particular care. The solution to this PDE is a function, and it can be cast as the solution to an optimization problem (by the same principle that states a physical system at rest is in the state that minimizes energy.) Thus, we must minimize some cost function (an energy function) over a space of functions. Spaces of functions are usually infinite dimensional, so we have to do something about that. The key step in FEM is to mesh the domain of the PDE (the airplane wing) into small elements (tetrahedrons for example), and to define a low-dimensional family of functions over this mesh. A trivial example is to allow the value of the function at each mesh vertex to be a variable, and to define the function at all other points through piecewise linear interpolation. If the energy function is quadratic (as is normally the case), minimizing it as a function of the vertex values boils down to one (very) large and structured linear system (each element only interacts with its immediate neighbors, so that the corresponding matrix has to be sparse.) Such systems are best solved with matrix-free solvers.

This idea of reducing an infinite dimensional optimization problem over a space of functions to a finite dimensional problem (apparent in our work with polynomial approximation, and also present in FEMs as described above) is also the root of all modern applications of (deep) neural networks in machine learning, which you have certainly heard about. There, the basic problem is as follows: given examples $(x_1, y_1), \ldots, (x_n, y_n)$ (say, x_i is an image, and y_i is a number which states whether this image represents a cat, a dog, ...), find a function f such that if we encounter a new image (an image we have never seen before), then $y = f(x)$ is a good indication of what the image contains (a cat, a dog, ...) This problem is called learning. At its heart, it is a function approximation problem if we assume that a "perfect" function f indeed exists (what else could we do?) Neural networks are a fancy

way of defining a finite-dimensional space of nonlinear functions. This space is parameterized by the weights of the neural connections. To learn, one optimizes the weights such that the neural network appropriately predicts the y_i's at least for the known x_i's. Thus, it is an optimization problem over a finite-dimensional space of functions. Of course, there is a lot more to it (for instance, it is of great importance to determine whether the obtained neural network actually learned to generalize, as opposed to just being able to parrot the known examples), but this is the gist of it.

Welcome to the world of numerical analysis!

Bibliography

[SM03] Endre Süli and David F. Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003.

[TBI97] Lloyd N. Trefethen and David Bau III. *Numerical linear algebra*, volume 50. SIAM, 1997.