CS 3430 S26: Scientific Computing
Assignment 3
Newton-Raphson, Random Linear Systems, Edge Detection

Vladimir Kulyukin
School of Computing
College of Engineering
Utah State University

January 24, 2026

Learning Objectives

1. Newton-Raphson, Zero Roots of Polynomials, Irrational Numbers (they are actually very reasonable).
2. Solving Linear Systems and experiencing brittleness under singular or ill-conditioned data.
3. Least Squares with `np.linalg.lstsq` as a resilient method for noisy/dirty data.
4. Matrix Algebra in Edge Detection: Sobel-like kernels, gradient magnitude, thresholding, and seeing linear algebra come alive on real images.

Introduction

In this assignment, we will get hands-on experience with three classic (and very practical!) scientific computing ideas: Newton-Raphson root finding, solving linear systems, and edge detection. The unifying theme is that scientific computing is about turning problems into mathematical objects (numbers, functions, vectors, matrices) and then computing with them reliably in code.

Along the way, we will see a recurring theme in computer science: clean mathematics meets dirty data. Some methods are powerful but brittle, while others (that are not as mathematically pretty) are more resilient when the data are noisy, inconsistent, or ill-conditioned, which happens in real life more often than we would like.

We will also play with irrational numbers. Despite the name, irrational numbers are actually quite reasonable: they just refuse to be written as ratios of integers. In other words, they are not irrational—they are simply independent-minded!

Finally, we will apply linear algebra to images. This is linear algebra live: when our code detects edges in a real image, matrices stop being abstract tables of numbers and start behaving like living computational objects.

Please review the Lecture 5 PDF and/or your notes before starting this assignment—your future self will recursively thank you in $O(1)$ time instead of $O(n)$ time.

Problem 1: Newton-Raphson (2 points)

The Newton-Raphson Algorithm (NRA) is applicable whenever we need to find a zero root of a polynomial function (e.g., a polynomial function $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0 x^0$). A value of

x is a zero root of $f(x)$ if $f(x) = 0$. NRA requires us to differentiate polynomial functions.

This algorithm generates a sequence of guesses $x_0, x_1, \ldots, x_n$ where each subsequent guess is usually a better approximation of a zero root of $f(x)$. There are two stopping conditions for NRA:

1. $|x_i - x_{i-1}| \leq \epsilon$, i.e., the absolute difference between two consecutive values of x is less than or equal to some small error value, e.g., $\epsilon = 0.0001$; and
2. we set the number of iterations and return x_n after n iterations of the algorithm.

The algorithm repeatedly executes the following formula:

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}, i > 0.$$

The initial guess (i.e., x_0) is typically obtained from graphing the function or generating a random guess and then observing the values of $f(x)$ at each subsequent guess.

The file `nra.py` contains the stubs of several static methods you will implement to find zero roots of polynomials with the NRA using NumPy and SymPy.

All methods take polynomials in the form of a Python text string such as `"x**2 - 2"`, `"x**3 - 2"`, or `"x**2 - x - 6"`. (Make sure you use `x**2`, not `x^2`.)

The `np_zr1(text, x0, num_iters=3)` and `sp_zr1(text, x0, num_iters=3)` methods take:

- a string `text` with a polynomial in variable `x`;
- the first approximation to a zero root `x0`;
- the number of iterations `num_iters`.

They run NRA for the specified number of iterations and return the float zero root approximation found after those iterations. The difference between the two methods is:

- `np_zr1` uses SymPy to parse the polynomial and compute the derivative, but uses NumPy floats during NRA iterations;
- `sp_zr1` uses SymPy to parse, differentiate, and perform the NRA iterations using SymPy numeric objects.

The `np_zr2(text, x0, delta=0.0001)` and `sp_zr2(text, x0, delta=0.0001)` methods take:

- a string `text` with a polynomial in variable `x`;
- the first approximation to a zero root `x0`;
- a tolerance `delta` that specifies how close two consecutive approximations must be.

These methods keep computing approximations until the absolute difference between two consecutive approximations is $\leq$ `delta`, and then return the final approximation.

Note on `@staticmethod`: In Python, the decorator `@staticmethod` means that a method belongs to the class (e.g., `nra` or `rls`) rather than to a particular object created from that class. In other words, you call it directly as `nra.np_zr1(...)` or `rls.solve_lin_sys(...)` (see Problem 2), without creating an instance such as `obj = nra()`. Static methods are a convenient way to group related functions into a single class without storing any extra state.

The file `nra.py` contains the following helper method for you to check how good a computed zero root approximation `zr` is:

```
def check_zr(text, zr):
    x, f = nra._parse_poly(text)
    f_np = sp.lambdify(x, f, modules="numpy")
    return np.allclose(float(f_np(float(zr))), 0.0)
```

This method takes a polynomial string `text` and a float value `zr` and returns `True` if $f($`zr`$)$ is sufficiently close to 0 (up to a small numerical error). In other words, it checks whether `zr` is a good approximation of a zero root of the polynomial.

The file `cs3430_s26_hw_3_prob_1_uts.py` contains unit tests you will run to test your implementations.

Important SymPy note: `subs`. In SymPy, the method `subs` means *substitute*. For example, suppose `f = x**2 - 2`. Then:

- `f.subs(x, 2)` produces $2^2 - 2 = 2$;
- `f.subs(x, 1.5)` produces $1.5^2 - 2 = 0.25$;
- `f.subs(x, -1)` produces $(-1)^2 - 2 = -1$.

Thus, `f.subs(x, value)` evaluates the symbolic expression `f` at a particular numeric value of `x`.

Irrational Numbers

We can use differentiation and the NRA to approximate the values of irrational numbers such as $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\sqrt{11}$, $\sqrt{13}$, ...

How? We observe that $\sqrt{2}$ is a zero root of $x^2 - 2$; $\sqrt{3}$ is a zero root of $x^2 - 3$; $\sqrt{5}$ is a zero root of $x^2 - 5$; $\sqrt{7}$ is a zero root of $x^2 - 7$; $\sqrt{11}$ is a zero root of $x^2 - 11$; $\sqrt{13}$ is a zero root of $x^2 - 13$, etc.

Thus, we can use NRA and a polynomial differentiation engine (SymPy in our case) to approximate irrational numbers. In this problem, you will implement:

- `compute_irrational_sqrt(n, x0, delta)`, which approximates $\sqrt{n}$ by finding a zero root of `x**2 - n`;
- `compute_irrational_cubic_root(n, x0, delta)`, which approximates $\sqrt[3]{n}$ by finding a zero root of `x**3 - n`.

The Famous $\sqrt{2}$

There is a story here, sad but inspiring. The ancient Greek mathematicians and astronomers (especially, the Pythagoreans) believed that the natural numbers (by the way, they did not consider 0 as a natural number) were the only true numbers, because they were "God-given" [1]. They believed that the rational numbers were ratios of natural numbers and were gapless, i.e., did not contain any breaks in them. In fact, the Pythagoreans viewed the rational numbers as a continuous "flow of quantity."

All was well until a member of the Pythagorean Brotherhood, long after the death of Pythagoras himself, decided to measure the length of the diagonal of a unit square. I should note that although the Pythagoreans called themselves the *Brotherhood*, women were completelly equal to men among them. In fact, Pythagoras' wife, Theano, became the leader of the Brotherhood after Pythagoras' death and wrote several important papers on various aspects of mathematics.

The Pythagorean scholar who decided to measure the length of the diagonal in the unit square knew the Pythagorean theorem, of course, and so the scholar reckoned that if h is the length of the diagonal, then $h^2 = 1 + 1$ and $h = \sqrt{2}$. Since the rationals were considered gapless, the unknown scholar further reckoned that $h = \sqrt{2} = m/n$, for some natural numbers m and n (recall that 0 was

not considered a natural number). Thus, since $\sqrt{2} = m/n$, one had to have $m^2 = 2n^2$. The scholar also knew that Euclid, by that time, had already shown that every natural number greater than 1 has a unique prime factorization (i.e., can be represented as a unique product of prime numbers). Since the scholar was squaring two natural numbers m^2 and n^2, every prime factor of m and n had to appear in the prime factorizations of m^2 and n^2 a even number of times. But then the number of times that 2 appears in $2n^2$ is odd. B-bbb-o-o-o-ooo-ooo-m-mmm!!! The mental detonation in that scholar's mind must have been loud! If $m^2 = 2n^2$ and every number has a unique prime factorization, the number of times 2 occurs in m^2 must be the same as the number of times it occurs in $2n^2$, but that, alas, was not the case. This famous argument is the foundation of the proof that $\sqrt{2}$ is not a rational number.

The Pythagoreans did not treat well the discovery that $\sqrt{2}$ was not rational. They called it $\alpha\lambda o\gamma o\sigma$ (alogos), which means "unspeakable" or "inexpressible." Some historians of mathematics believe that this was the reason why numbers such as $\sqrt{2}$ were later called *irrational* (unreasonable). Note a fine touch of neurolinguistic conditioning in this term: $\sqrt{2}$ was discovered by human reason (ratio) yet is called unreasonable (irrational).

The Pythagoreans took an oath to never share this knowledge with people outside of their Brotherhood. Anyone who would reveal this knowledge to the public would be put to death. A member the Brotherhood (we don't know if it was the same member who proved the irrationality of $\sqrt{2}$ or a different person) revealed this knowledge to the public. A Greek philosopher of the 5-th century B.C.E writes that the person who "fortuitously revealed this aspect of the living things" to the world perished in a shipwreck. Other historians of mathematics commenting on this episode write that that person was put to death by drowning.

This story is sad, because a mathematician gave up his/her life for speaking the truth. However, it is inspiring and, indeed, "fortuitous" to all of us, because centuries later the Greak astronomer and mathematician Eudoxus, a student of Plato, offered the first (currently known to historians of mathematics) definition of irrational numbers, thus foreshadowing the work of the great Georg Cantor in the second half of the 19-th century. Cantor showed that irrational numbers are much more numerous and common than rational numbers (and, consequently, as one may say, more reasonable than the numbers we actually *rational*, i.e., *reasonable*). There is little irrationality in irrational numbers.

Problem 2: Random Linear Systems (1 point)

In scientific computing, one of the most common computational tasks is solving a linear system

$$Ax = b,$$

where:

- A is a matrix of coefficients (constraints);
- x is a vector of unknowns;
- b is a vector of outcomes (measurements).

When A is a square matrix ($n \times n$) and is invertible (i.e., not singular), the system $Ax = b$ has a unique solution. In this problem, you will solve many such systems using NumPy.

In *real* scientific computing, the data in A and b are often "dirty": they may come from sensors, measurements, experiments, or manual data entry (a sigh, then a deep sigh). Such data may contain noise, errors, or inconsistencies. As a result, A may be singular or close to singular. This is sometimes called *ill-conditioning*. The practical meaning of ill-conditioning is that even a small

change in the input data can produce a very large change in the computed solution. In other words, the transformation collapses dimension, so it cannot be inverted.

The file `rls.py` (Random Linear Systems) contains the stubs of several static methods you will implement. These methods will allow you to experience both:

- the power of solving linear systems efficiently;
- the brittleness of exact solvers when data are noisy or singular.

In `rls.py`, you will implement:

1) `solve_lin_sys(A, b)`
This method takes a square matrix `A` and a column vector `b` and solves the linear system $Ax = b$ using NumPy's direct solver `np.linalg.solve`. If a unique solution exists, it returns the solution vector `x` as an `n x 1` NumPy array.

2) `solve_rand_lin_sys(n, lower, upper, seed)`
This method generates a random linear system of size n:

- `A` is an `n x n` matrix with random real values in the range `[lower, upper]`;
- `b` is an `n x 1` vector with random real values in the range `[lower, upper]`.

It then solves the generated system and returns the solution vector `x`. If the optional argument `seed` is given, the random system is reproducible (i.e., the same `A` and `b` will be generated every time for that seed). That said, I have seen Python implementations where this promise is not honored. So, take it with a grain of salt.

3) `solve_lin_sys_ls(A, b)`
Sometimes a linear system has no exact solution, because the data are noisy, inconsistent, or the matrix is singular. In such cases, we can still compute the *best approximate* solution using least squares:

$$\min_x \|Ax - b\|_2.$$

This method uses NumPy's least-squares solver `np.linalg.lstsq` and returns four objects:

- `x`: the least-squares solution vector (best approximate solution);
- `residuals`: how much error remains after fitting;
- `rank`: the rank of A (how many independent constraints exist);
- `s`: the singular values of A (how close A is to collapsing dimension – recall our discussion of determinants in Lecture 5!).

4) `safe_solve_lin_sys(A, b)`
In real scientific computing, we often want our programs to be *resilient*. This method attempts to solve a system with the direct solver first. If that succeeds, it returns the result. If it fails (because the direct solver is brittle), it automatically falls back to least squares.

The method returns a 5-tuple:

```
(method, x, residuals, rank, s)
```

where `method` is either `"solve"` or `"lstsq"`, depending on which method was used.

The file `cs3430_s26_hw_3_prob_2_uts.py` contains unit tests you will run to test your implementations. These tests will include solving small systems (such as 2×2 and 3×3) as well as very large systems (such as 500×500). You should expect large systems to run quickly if your code is correct, because NumPy is highly optimized for matrix computations.

Problem 3: Edge Detection (2 points)

In this problem, we will play with a small but powerful example of scientific computing: *edge detection* in images. This is a beautiful place where linear algebra comes alive.

A grayscale image can be represented as a matrix:

$$I[r, c] \in [0, 255],$$

where each entry $I[r, c]$ is the intensity (brightness) of pixel (r, c):

- 0 means black,
- 255 means white.

Edges are locations in an image where intensity changes rapidly. A classic way to detect edges is to apply a small 3×3 **kernel** (filter) at every pixel location and compute a weighted sum of pixels. This is a structured linear algebra computation.

In the Lecture 5 PDF, I define two Sobel-like edge detection kernels:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The kernel K_x responds strongly to left-to-right (horizontal) intensity changes and therefore detects *vertical* edges, producing the response matrix G_x. The kernel K_y responds strongly to top-to-bottom (vertical) intensity changes and therefore detects *horizontal* edges, producing the response matrix G_y. Intuitively, if the intensity changes sharply as we move top-to-bottom (a vertical direction), then the pixels where this happens typically form a *horizontal* edge (a horizontal boundary). Likewise, if the intensity changes sharply as we move left-to-right (a horizontal direction), then the pixels where this happens typically form a *vertical* edge (a vertical boundary). These responses are computed by sliding the kernel over the image and performing a weighted sum of pixel values at each location. This is linear algebra live! Image processing is what makes me fall in love with linear algebra deeper and deeper.

The file `edge.py` contains the stubs of several static methods you will implement:

1) `sobel_ky_response(I)`
This method takes a grayscale image matrix `I` and computes the Sobel-like response matrix `Gy` by applying the kernel K_y at every interior pixel location. The output is a matrix `Gy` of the same shape as `I`. Border pixels are left as 0.

2) `grad_magnitude(Gx, Gy)`
This method computes the gradient magnitude at each pixel location:

$$|G| = \sqrt{G_x^2 + G_y^2}.$$

The matrix $|G|$ combines horizontal and vertical edge strengths into a single response matrix.

3) `binary_edge_map_from_gradmag(Gx, Gy, T)`
This method computes:

- the gradient magnitude matrix `Gmag`;
- a binary edge map `E` such that:

$$E[r, c] = \begin{cases} 255 & \text{if } |G[r, c]| \geq T, \\ 0 & \text{otherwise.} \end{cases}$$

The threshold `T` controls how strong an intensity change must be before we call it an edge. If `T` is too low, too many pixels become edges. If `T` is too high, you may miss weaker edges.

The file `cs3430_s26_hw_3_prob_3_uts.py` contains unit tests you will run to test your implementations. These unit tests enable you to experiment with both:

- **synthetic images** (simple step-edge patterns designed for clarity);
- **real images** (photographs provided in the `imgs/` directory).

What To Submit

1. cs3430_s26_hw_3.zip with your solutions in nra.py (Problem 1), rls.py (Problem 2), and edge.py (Problem 3).
2. Please do not submit the images in `imgs/` or the unit test files. When we grade your submissions we will drop `imgs/` and the unit tests in your subfolder on our end and then run the unit tests. These omissions will significantly reduce the size of your zip. Canvas will not complain about your file sizes.

Happy Hacking! Enjoy Irrational Numbers and Live Linear Algebra!

References

1. L. Goldstein, D. Lay, D. Schneider, N. Asmar. *Calculus and its Applications.*
2. E. Burger. *Zero to Infinity: A History of Numbers.* The Great Courses Publishing, 2007.
3. www.sympy.org.