Jose Manuel Ossorio
Marco Antonio Pérez
Johan Camilo Cortés

Engineering method

Phase 1: Identifying the problem:

- Description of the problem context:
  - The developers using the Java programming language require the implementation of two algorithms to find the real roots of a polynomial
  - Currently, the Java 11 JDK does not include an algorithm to find the real roots of a polynomial
  - The implementation of the algorithms will facilitate the work of engineering students
- Description of the problem:
  - Oracle corporation requires the development of a software (implementing two algorithms) which properly finds the roots of a polynomial function up to the tenth power.
- Functional Requirements:
  - The software must:
    - R1: allow the user to enter the polynomial through the use of a graphic interface
    - R2: generate random polynomials up to the tenth power using the GUI
    - R3: use the randomly generated polynomials as input for the algorithms
- Non-functional requirements:
  - The software must:
    - Implement two efficient algorithms.
    - Use JavaFX to create the GUI

Phase 2: Compilation of necessary information:

Sources:

Precálculo, Matemáticas para el cálculo. J. STEWART

Definitions

- Function: a function *f* is a rule that assigns to each element x from set A exactly one element, named *f(x)*, from set B. Functions describe the dependency of one quantity to another.
- Order of a function: the order of a function refers to the largest exponent of sum of exponents in any of its monomyals.
- Polynomial function: a function which is defined by an expression including polynomials is said to be a polynomial function. Therefore, a polynomial function of order *n* follows: $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$.

- Roots of a polynomial function: the roots of a polynomial are numbers which make the function $P(x)$ equal to zero.

Key elements in the solution of the problem

- JavaFX:
  - ChoiceBox
  - Button
  - TextField
  - ImageView

- Java libraries:
  - Math library
  - Random library
  - JScience

In order to solve the problem, we researched different approaches to find the better and more efficient ones:

- Newton's method: helps finding successively better approximations to the roots of a function.
- Secant method
- Inverse interpolation
- Brent's method
- Bisection method
- False position
- Steffensen's method
- Bairstow's method
- Aberth method
- Durand–Kerner method

Phase 3: Creative solutions

We researched methods to find the roots of a polynomial in the previous phase by brainstorming and in this phase we will describe them very briefly. We will include some of our thoughts into the description of the methods regarding why we might think using this method is a good idea or not.

- Newton's method: It begins with an initial function and a first guess to a root of the function. If the function satisfies the assumptions made in the derivation of the formula and the initial guess is close, then we can find a better approximation to the root (x_1) by operating: $x_0 - \frac{f(x_0)}{f'(x_0)}$. the process repeats itself until a sufficiently accurate approximation is found. This method could be tricky because it requires the derivative of the function in order to work, but since we are only using polynomial it is not that hard to differentiate the function. For this method we gathered information from these sources:

  - https://en.wikipedia.org/wiki/Newton%27s_method
  - http://mathworld.wolfram.com/NewtonsMethod.html

- Secant method: This method begins with two x_1 and x_2 values using the following formula $x_3 = x_1 - \frac{f(x_1)(x_2 - x_1)}{f(x_2) - f(x_1)}$ we can obtain a x_3 value which is a nearby value to the root of the polynomial. We continue doing this action until the absolute value of the error, which is calculated with the following formula: $error = \frac{x_3 - x_2}{x_2} * 100$, is less than 1%.
  Sources:
  https://es.wikipedia.org/wiki/M%C3%A9todo_de_la_secante
  https://matematica.laguia2000.com/general/metodo-de-la-secante-2

- Bairstow's method: it aims to use Newton's method to adjust the coefficients *u* and *v* in the quadratic polynomial $x^2 + ux + v$. This adjustment is made until the roots of said polynomial are the roots of the original function.
  Sources:
  https://en.wikipedia.org/wiki/Bairstow%27s_method
  https://nptel.ac.in/courses/122104019/numerical-analysis/Rathish-kumar/ratish-1/f3node9.html

- Inverse Interpolation: it uses quadratic interpolation to approximate the inverse of f. It is important for another possible method to find roots of polynomial, the Brent's method. Polynomial interpolation is the interpolation of a given data set by the polynomial of lowest possible degree that passes through the points of the dataset. Therefore, with this algorithm we can find the zeros of a polynomial by computing $f^{-1}(0)$, where $x = f^{-1}(y)$ is a polynomial that interpolates a table of $(f^{-1}(x),\ (x))$ values.

  Sources:
  http://www.sze.hu/~lotfi/Interpolation-2.pdf
  https://en.wikipedia.org/wiki/Polynomial_interpolation
  https://en.wikipedia.org/wiki/Inverse_quadratic_interpolation

- Brent's Method: it combines the secant method, the bisection method and inverse quadratic interpolation.
  Sources:
  https://en.wikipedia.org/wiki/Brent%27s_method

- Bisection method: This method takes an interval of numbers in the x axis with an x_i value as the lower limit and an x_f value as the upper limit of the interval. Next, it calculates the value of half the interval $x_h = \frac{x_f + x_i}{2}$. Then, it multiplies the $f(x_i)$ value and the $f(x_f)$ value to see the sign of that operation. If the sign of multiplying those values is negative, it means the root is closer to the lower limit, so we assign the value of half the interval to the value of the upper limit. If on the contrary, the value of the multiplication is positive, it means the root is closer to

the upper limit, then, we assign the value of half the interval to the value of the lower limit. This method continues until the difference between both limits is the minimum value (absolute value less than 1).
Sources:
https://tecdigital.tec.ac.cr/revistamatematica/HERRAmInternet/ecuaexecl/node4.html

- Durand–Kerner method: This method allows the user to obtain all real and complex roots of the polynomial expression evaluated. The method is better understood by following an example. If, for example, one gets a polynomial defined as:
$f(x) = ax^3 + bx^2 + cx + d$, for said polynomial, the term *a* has to be 1, therefore, if this isn't the case, every expression shall be divided by *a* in order for the method to work. Afterwards one must pick values for *P*, *Q*, *R* (complex) avoiding roots of unity.Finally, the values can be calculated by iterating the following calculations until the values stop variating (defined by a premade choice on precision):
  - $p_n = p_{n-1} - \frac{f(p_{n-1})}{(p_{n-1}-q_{n-1})(p_{n-1}-r_{n-1})}$;
  - $q_n = q_{n-1} - \frac{f(q_{n-1})}{(q_{n-1}-p_{n-1})(q_{n-1}-r_{n-1})}$;
  - $r_n = r_{n-1} - \frac{f(r_{n-1})}{(r_{n-1}-q_{n-1})(r_{n-1}-q_{n-1})}$.

  https://en.wikipedia.org/wiki/Durand%E2%80%93Kerner_method

- Aberth's method: this is a method that can find all the roots of a polynomial at the same time and it converges cubically, an improvement from Durand-Kerner's.
  Source:
  https://en.wikipedia.org/wiki/Aberth_method

Phase 4: Transition to preliminary design

We have decided to rule out the following algorithms due to their advanced complexity of implementation and comprehension:

- Bisection
- Brent's method
- Inverse interpolation

We have selected the following algorithms as the most viable:

- Newton's method: this method is very easy to understand and implement since it only requires the derivative of the polynomial. However, it will only find one root of the polynomial, which is not what we need in this case.
- Secant method: as the previous method, this method will only find one root, therefore we won't use it, but it is as simple as Newton's in terms of comprehension and understanding
- Aberth method: this method is very efficient and easy to implement due to the fact that we will be using a library called JScience. This library will allow us to manipulate complex numbers required to implement this algorithm.
- Durand–Kerner method: very similar to the previous one, it will require the use of complex numbers in order to find all the real and imaginary roots of the polynomials.

- Bairstow's method: this method uses Newton's method with some improvements and changes and it can actually find all the roots of the polynomial.

However, we have finally decided to implement only Durand-Kerner method and Aberth's method because Newton and Secant method only give one root of the polynomial and Bairstow's method represented a bigger challenge in terms of implementing the algorithm and understanding the math behind it.

Phase 5: Evaluating and selecting the best solution

Criteria:

- Criterion A: satisfactory solution of the problem
  [1] The method does not solve the problem satisfactorily
  [2] The method partially solves the problem
  [3] The method solves the problem satisfactorily

- Criterion B: complexity of the implementation of the algorithm
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex

- Criterion C: complexity of the understanding of the algorithm
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex

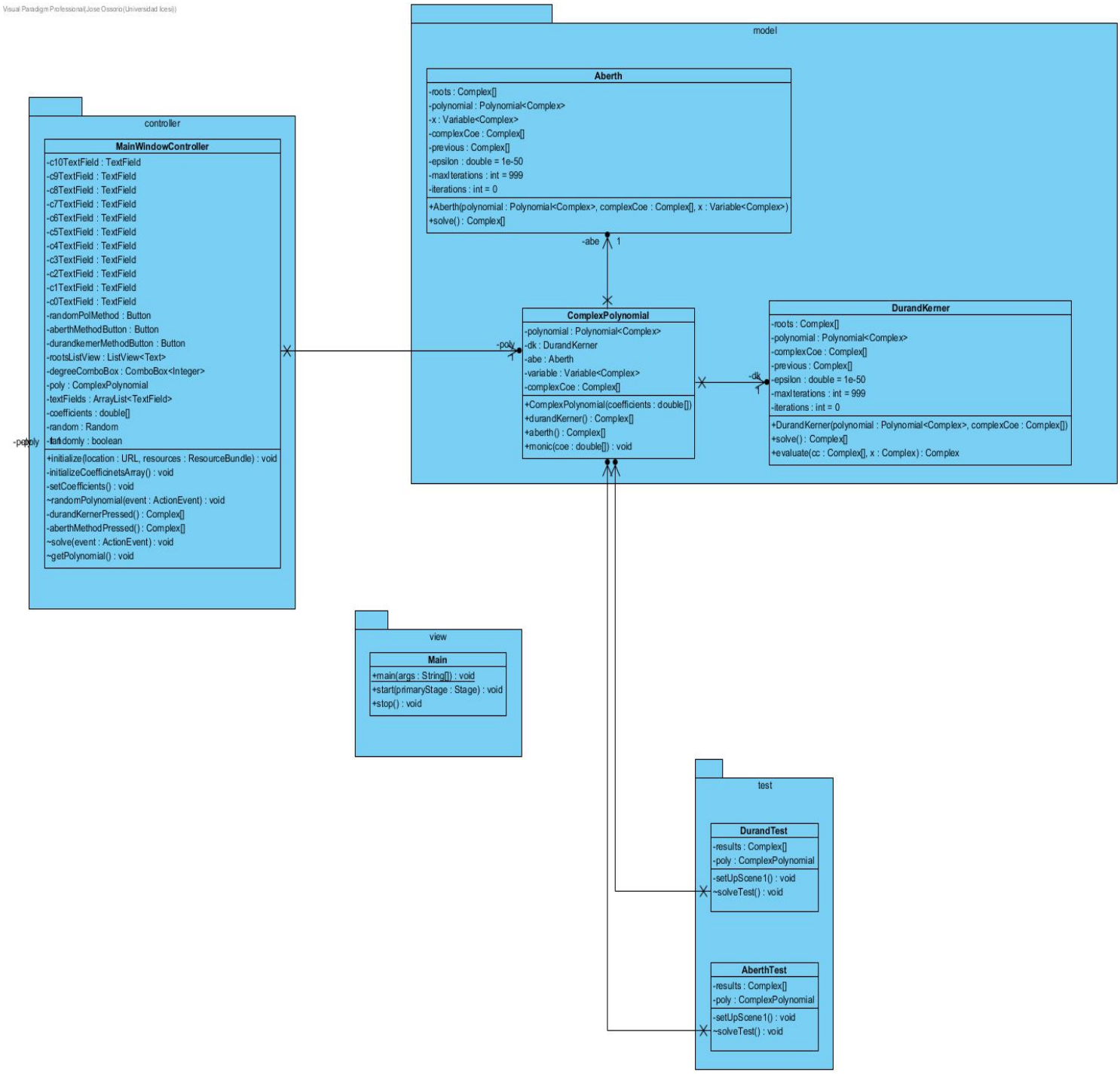
- Criterion D: time efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient

- Criterion E: space efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient

Algorithm	Criterion A	Criterion B	Criterion C	Criterion D	Criterion E	Total
Bisection	1	4	4	3	4	16
Brent's method	2	1	2	3	4	12
Inverse interpolation	2	3	3	3	4	15
Newton's	1	4	4	3	4	16
Secant	1	4	4	3	4	16
Aberth	3	4	3	3	4	17
Durand-Kerner	3	4	3	3	4	17
Bairstow's	3	2	2	3	3	14

Phase 6: Preparation of reports and specifications

- **Class Diagram:**



The following pseudocode will only represent the algorithms in themselves, therefore, alone they most likely won't solve the problem as they may need auxiliary methods but they are an accurate representation of the algorithms.

- **Pseudocode for Durand-Kerner algorithm:**

```
durandKerner(Array a0, Array a1, Polynomial poly)
    epsilon ← 1e-50
    finished ← true
    result ← 1
    while(!finished)
        from i = 0 to a0.length
            from j = 0 to a0.length
                if(i != j)
                    result ← (a0[i] - a0[j])*result
            a1[i] ← a0[i] - ((poly.evaluate(a[i]))/result)
    from i = 0 to a1.length
        if(Math.abs(a1[i]-a0[i] < epsilon)
            finished ← true
    return a1
```

- **Pseudocode for Aberth's algorithm:**

Note: since Aberth's algorithm needs to differentiate the polynomial, it will output NaN and NaN*i as the zeroes for a function of degree 1.

```
aberth(Array a0, Array a1, Polynomial poly)
    epsilon ← 1e-50
    finished ← true
    numerator ← 1
    denominator ← 1
    derivPol ← poly.differentiate("x")
    while(!finished)
        from i = 0 to a0.length
            from j = 0 to a0.length
                if(i != j)
                    numerator ← (poly.evaluate(a0[i])/(derivPol.evaluate(a0[i]))
                    denominator ← (1/(a0[i] - a0[j])) + denominator
            denominator ← (1 - numerator) * denominator
            a1[i] ← a0[i] - (numerator / denominator)
    from i = 0 to a1.length
        if(Math.abs(a1[i]-a0[i] < epsilon)
            finished ← true
    return a1
```

- **Unitary tests design:**

Method	Scene	Input	Output
durandKerner()	An array of coefficients is created for which the method will find the roots and then evaluate them in the function to verify they are indeed zeroes	**None**	A number smaller than 1e-5, which will represent a zero
aberth()	An array of coefficients is created for which the method will find the roots and then evaluate them in the function to verify they are indeed zeroes	**None**	A number smaller than 1e-5, which will represent a zero

The following time complexity analysis will assume there is no maximum number of iterations assuming the programmer would like only the comparison with epsilon to define if the loop should stop.

- **Time-complexity analysis for Durand-Kerner algorithm:**

Note: n = previous.length, k = roots.length

#	solve()	coe	# executions
1	boolean finished = false;	c1	1
2	Complex result;	c2	1
3	int iterations = 0;	c3	1
4	while(!finished) {	c4	k +1
5	for(int i = 0; i < previous.length; i++) {	c5	(n + 1)*k
6	result = Complex.ONE;	c6	((n + 1)*k) - k
7	for(int j = 0; j < previous.length; j++) {	c7	(n+1)*(((n + 1)*k) - k)
8	if(i != j) {	c8	(n+1)*(((n + 1)*k) - k)-(((n + 1)*k) - k)
9	result = previous[i].minus(previous[j]).times(result);	c9	(n+1)*(((n + 1)*k) - k)-(((n + 1)*k) - k)
10	roots[i] = previous[i].minus(evaluate(complexCoe, previous[i]).divide(result));	c10	((n + 1)*k) - k
11	for(int i = 0; i < roots.length; i++) {	c11	(k+1)*k
12	if((Math.abs(roots[i].minus(previous[i]).getReal()) < epsilon && Math.abs(roots[i].minus(previous[i]).getImaginary()) < epsilon) || !(iterations < maxIterations)) {	c12	((k+1)*k) - k
13	finished = true;	c13	1
14	iterations++;	c14	k
15	System.arraycopy(roots, 0, previous, 0, roots.length);	c15	k
16	return roots;	c16	1

T(n) = 1+1+1+k+1+nk+k+nk+n*n*k+nk+n*n*k+nk-nk+n*n*k+nk-nk+nk+k*k+k+k*k+1+k+k+1
*Since n and k represent the same quantity, then
T(n) = 6 + 5n + 4n^2 + 3(n^3) + 2n^2 = O(n^3)

- **Time-complexity analysis for Aberth's algorithm:**

#	solve()	coe	# executions
1	boolean finished = false;	c1	1
2	Complex numerator;	c2	1
3	Complex denominator;	c3	1
4	int iterations = 0;	c4	1
5	while(!finished) {	c5	k + 1
6	for(int i = 0; i < previous.length; i++) {	c6	(n + 1)*k
7	numerator = Complex.ONE;	c7	((n + 1)*k) - k
8	denominator = Complex.ONE;	c8	((n + 1)*k) - k
9	for(int j = 0; j < previous.length; j++) {	c9	(n + 1)*(((n + 1)*k) - k)
10	if(i != j) {	c10	(n + 1)*(((n + 1)*k) - k) - (((n + 1)*k) - k)
11	numerator = polynomial.evaluate(previous[i]).divide(polynomial.differentiate(x).evaluate(previous[i]));	c11	(n + 1)*(((n + 1)*k) - k) - (((n + 1)*k) - k)
12	denominator = ((Complex.ONE.divide(previous[i].minus(previous[j]))).plus(denominator));	c12	(n + 1)*(((n + 1)*k) - k) - (((n + 1)*k) - k)
13	denominator = Complex.ONE.minus(numerator).times(denominator);	c13	(n + 1)*(((n + 1)*k) - k)
14	roots[i] = previous[i].minus(numerator.divide(denominator));	c14	(n + 1)*(((n + 1)*k) - k)
15	for(int i = 0; i < roots.length; i++) {	c15	(k + 1)*k
16	if((Math.abs(roots[i].minus(previous[i]).getReal()) < epsilon && Math.abs(roots[i].minus(previous[i]).getImaginary()) < epsilon) || !(iterations < maxIterations)) {	c16	((k + 1)*k) - k
17	finished = true;	c17	1
18	iterations++;	c18	k
19	System.arraycopy(roots, 0, previous, 0, roots.length);	c19	k
20	return roots;	c20	1

T(n) = 1 + 1 + 1 + 1 + k + 1 + kn + k + kn + kn + kn*n + kn + kn + kn*n + kn + kn*n + kn + kn*n + kn + kn*n + kn + kn + kn*n + kn + kn + k*k + k + k*k + 1
*Since n and k represent the same quantity, then
T(n) = 6 + 3n + 12n^2 + 6n^3 + 2n^2 = O(n^3)

- **Space-complexity analysis for Durand-Kerner algorithm:**

Type	Variable	Size of the atomic value	Amount of atomic values
Input	N/A	N/A	N/A
Auxiliary	result		n
	iterations	32 bits	1
	finished	2 bits	1
	i	32 bits	1
	j	32 bits	1
Output	roots		n

Total space complexity = n + 1 + 1 + 1 + 1 + n = 2n + 4 = O(n)

- **Space-complexity analysis for Aberth's algorithm:**

Type	Variable	Size of the atomic value	Amount of atomic values
Input	None	None	None
Auxiliary	finished	2 bits	1
	numerator		1
	denominator		1
	iterations	32 bits	1
	i	32 bits	1
	j	32 bits	1
	i	32 bits	1
Output	roots		n

Total space complexity = 1+1+1+1+1+1+1+n = 7+n = O(n)