Implementing Jenkins Traub algorithm with inverse Power Iteration

Asked 2 days ago Modified yesterday Viewed 40 times

1

The Wikipedia Article from the Jenkins-Traub iteration suggests, that the steps can also be implemented with the three different forms of inverse vector iteration. It suggests a matrix to perform this iteration. My question is how can one create this matrix, when the roots are unkown. Shouldn't it therefore be impossible to create the i.e. the polynomial P_1. Moreover it says coefficient matrix, but the Ps are polynomials? Also it is not clear which starting vectors and shifts to use, I suppose s_lambda. Furthermore there is an explanation in the original paper from jenkins and traub which assumes an inverse iteration with the coefficient matrix of the polynomial. I tried implementing their approach but so far the results don't add up with the results from the original algorithm. An implementation is i.e. here.

So far I have only the following code for the three stages (not including normalization):

```
def stage_1(A,h_lambda,max_iteration):
    h_bar_lambda = h_lambda / h_lambda[0]
    for _ in range(max_iteration):
      h_bar_lambda = A @ h_bar_lambda
    h_lambda = h_bar_lambda
    return h_lambda

def stage_2(A,s_lambda,h_bar_lambda,dim,max_iteration):
  for _ in range(max_iteration):
    h_bar_lambda = np.linalg.inv(A-s_lambda[i]*np.eye(dim)) @ h_bar_lambda
  h_lambda = h_bar_lambda
  return h_lambda

def stage_3(A,s_lambda,h_bar_lambda,dim,max_iteration):
  for _ in range(max_iteration):
    h_bar_lambda = np.linalg.inv(A-s_lambda[i]*np.eye(dim)) @ h_bar_lambda
    s_lambda = np.array([rayleigh_s_lambda(A,h_bar_lambda,s_lambda),0,0,0,0])
  h_lambda = h_bar_lambda
  return h_lambda,s_lambda

def rayleigh_s_lambda(A,v,s_lambda):
   return s_lambda.T @ (A @v) / (s_lambda.T @ v)
```

Stage 3 is mostly oriented by the original paper from jenkins and traub.

python math matrix numerical-methods eigenvalue Edit tags

Share Edit Follow Close Flag

asked 2 days ago

Ragon
113 3

New contributor

Sorted by:

1 Answer

Reset to default

Date modified (newest first)

2

Polynomial root finders using deflation to find a full factorization or root set are known to be more, or at all, stable if they find and process the smaller roots first.

The roots of a polynomial are the eigenvalues of its companion matrix `A` which is the coordinate representation of the linear operator `M_X` in the monomial basis.

The power iteration finds the largest eigenvalues first. To find small eigenvalues first one has to use the inverse power iteration (as in stage 1). In implementations, this should not use the inverse matrix, but a linear system solver based on some matrix factorization.

The convergence of the inverse power iteration can be accelerated using shifts of the spectrum of the original matrix so that the eigenvalue of interest is shifted close to zero. The shift is implemented by subtracting a multiple of the identity matrix. It makes thus sense to use the current best guess for the eigenvalue as shift value (as in stage 3). Due to the cost of matrix factorizations it is also a valid decision to keep the shift value constant over a medium number of steps (as in stage 2).

The main insight of Jenkins and Traub was that one can implement the shifted inverse power iteration on the companion matrix without any matrix operations at all. Going back to the polynomial as polynomial=finite sequence, the same results can be achieved using one or two long divisions by linear factors, a.k.a. Horner schemes (not invented by Horner, Ruffini or Holdred (?), but older folklore even then).

So in detail

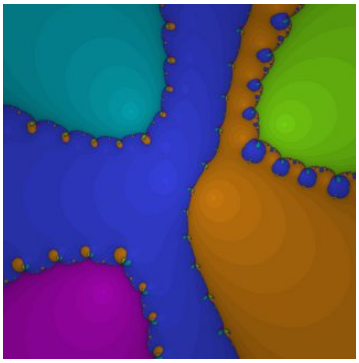
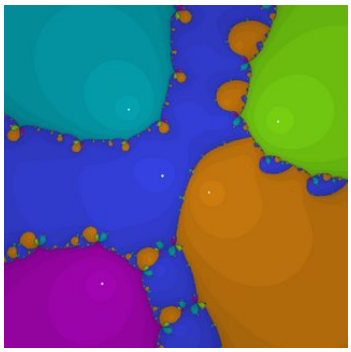
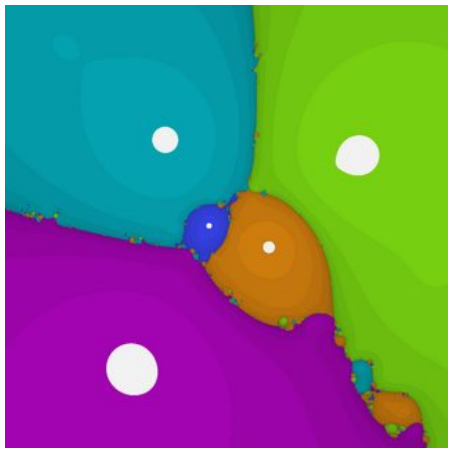
> how can one create this matrix, when the roots are unknown.

The matrix is just the companion matrix as written. Any equations containing roots address the (hypothetical) state at the end of the computation, like saying if `r` is a root, `P(r)=0`, then one can split off a linear factor, `P(x)=(x-r)*P_1(x)`. The Horner-Ruffini-Lagrange-Newton scheme or long division (with remainder) allows to compute such a factorization. If not done directly at the root, it also computes the residual, that is, the polynomial value, `P(x)=(x-a)*Q(x,a)+P(a)`.

If you want to directly work with the matrix, the Horner contemplations are irrelevant.

> Shouldn't it therefore be impossible to create the i.e. the polynomial P_1

Yes, yes it is. Knowing `P_1` is the same as knowing the root, which is the objective, not the initial state of the method. The idea of the iteration in Jenkins-Traub is that knowing approximations of `P_1` is likewise equivalent to knowing approximations of one root. Newton, Halley, Durand-Kerner etc. are based on improving the linear factor, that is, the root directly. Jenkins-Traub is based on improving the other polynomial factor and found that it improves reliability.

Newton	Halley	Jenkins-Traub stage 3
		

Fatou/Julia sets for the convergence of the first root using the Newton, Halley and Jenkins-Traub stage 3 iterations. Color brightness steps correspond to the number of iterations to reach the white region around the root. Note that one Newton step corresponds to two polynomial evaluation, while the other have 3 evaluations per step.

> Moreover it says coefficient matrix, but the Ps are polynomials?

If you know a better name, ... It is the matrix expressing a linear operator in a basis, so the entries are coefficients, coordinates, ...? It is based on the coefficients of `P`.

> Also it is not clear which starting vectors and shifts to use, I suppose s_lambda.

That is the purpose of stage 1 and stage 2. Stage 1 computes an initial polynomial factor starting with `H=P'`. With the unshifted inverse power iteration this converges, slowly, to the polynomial factor in the factorization of the smallest root. This is not very reliable as there might be multiple roots of similar small magnitude. The small random shift of stage 2 is less a root approximation and serves more to break the symmetry that might exist at the end of stage 1. The tendency is to select and emphasize the small root that is closest to the shift value. But such a root might not exist in a meaningful sense.

Anyway, at the end of stage 2 the expectation is that the polynomial `H` represents a good factorization, that is, that the (approximately) complementary linear factor contains a good root approximation that can be used in the fully shifted stage 3.

Mathematics can be terse, especially if a few symbols stand for a large amount of data or some complex algorithm. I rewrote this WP article with the aim to contain at least one complete algorithm (CPOLY is rather straight-forward, RPOLY has much more mysterious details) with all relevant formulas and ideas. Expanding this to didactic standards would give a text that is longer than the original, which contradicts WP being an encyclopedia. I'm open to improvements that do not overly increase the size of the article. But that should be discussed on the talk page of the WP article.

Share Edit Follow Flag edited yesterday answered 2 days ago

Lutz Lehmann
25.1k 2 22 51

Thank you for your extensive response! As far as I understand it I use a guess alpha, calculate P_0,...,P_n-1, then build the coefficient matrix and use it to iterate, resulting in a better guess of alpha? – Ragon yesterday

1 P_0,...,P_(n-1) are the coefficient of the polynomial P(X) = X^n + P_(n-1)X^(n-1)+...+P_1X+P_0. They are given, either as global input or as the cofactor to the linear factor of the last root. // It is from the original paper, but perhaps not that fortunate to use lambda for the integer iteration counter. There is no deeper meaning to that letter, one could equally well take k for the counter. – Lutz Lehmann yesterday

1 The objects H play the role of the iterated vectors on the (inverse) power iteration, the objects H_bar are the normalized vectors. Here normalization is setting the highest-degree coefficient to 1. So from H_bar[k] you compute H[k+1] and the normalization of that to H_bar[k+1] gives as factor the offset to the shift for the current root estimate/approximation. – Lutz Lehmann yesterday

1 I do not see from where you get this idea. The companion matrix is a (sparse) matrix of complex numbers. If you do inverse power iteration on it, then there are no further polynomials involved. The remaining connection is that the vector and the H polynomials are both finite sequences of complex numbers of a fixed length. As such sequences both interpretations of the algorithm should give identical results. – Lutz Lehmann yesterday

1 Yes, that is bad, using the same notation to express two different ideas in the same place. I'll change the coefficients to a_k as they were declared on-top. – Lutz Lehmann yesterday