CALCULATING THE MAXIMUM MODULUS OF A POLYNOMIAL USING STEČKIN'S LEMMA

J. J. GREEN*

Abstract. An algorithm for the calculation of the maximum modulus of a complex polynomial on the unit disc is described and evaluated. The algorithm is based on a lemma of Stečkin.

Key words. Maximum modulus, polynomial, Stečkin's lemma, von Neumann's inequality, spectral radius.

AMS subject classification. 65K10

1. Introduction.

If p is a polynomial in one variable then the *maximum modulus* of p is

$$\|p\|_\infty := \sup\{|p(z)| : z \in \mathbf{C},\ |z| \le 1\},$$

a quantity which arises in a number of interesting results in mathematics. For example, *von Neumann's inequality* [6, Chapter 6] asserts that for any contraction T on a Hilbert space the inequality

$$\|p(T)\| \le \|p\|_\infty$$

obtains. It is also interesting to note that $\|p\|_\infty$ is the *spectral radius* of the polynomial p considered as an element of the convolution (Banach) algebra $\ell^1(\mathbf{N})$. In this note we address the problem of calculating good bounds on $\|p\|_\infty$ in a reasonable amount of time.

Since $\|p\|_\infty$ is attained by $|p|$ on the boundary of the unit disc our objective is to find bounds on

$$\max_{t\in[0,2\pi]} \left|p(e^{it})\right|$$

and since our calculations are simplified if we work with the square of $|p|$, we write

$$q(t) := \left|p(e^{it})\right|^2 = p(e^{it})\overline{p(e^{it})}.$$

2. The Subdivision Algorithm.

The problem of finding the global maximizer of a function f on a subset I of $\mathbf{R}^n$ has been addressed by a number of authors. One popular approach is that of subdivision: one divides I into a series of subsets and those that are guaranteed not to contain the global maximizer are rejected. The remaining subsets are further tested, subdivided and so on. Eventually one obtains a set which contains all of the global maximizers and that is specified to some required degree of accuracy.

One example is the algorithm described in [2] and attributed there to Hansen [3]. The algorithm uses rejection criteria based on the methods of *interval analysis* and will obtain the global minimum and minimizers of any twice continuously differentiable function $[a, b] \to \mathbf{R}$ that is the finite algebraic combination of elementary functions (in a sense made precise in [2]). Another approach is adopted by Kavvadias and Vrahatis

*22 Wansfell Rd., Grimesthorpe, Sheffield, UK.

in [4] where a rejection criterion based on deep results in topological degree theory is used.

The performance of the subdivision method obviously depends on the quality of the rejection criteria, quality loosely being a combination of inexpensive computation and comprehensiveness (i.e. the rejection of as many subsets as possible). It seems inevitable that there will be some trade-off between these requirements, and so a balance to be found between them.

In the case that the objective function is as simple as q (and so inexpensive to calculate) rejection criteria that strike the balance with emphasis on computational economy are feasible, and may offer better performance. The remainder of this note describes such a criterion.

3. Bounds from Stečkin's Lemma. For the non-negative trigonometric polynomial q it is not difficult to obtain criteria that guarantee that an interval $[a, b]$ does not contain the global maximizer. The obvious application of Taylor's theorem gives good bounds on q in the interval provided that we know the value of q (and possibly some of its derivatives) at the midpoint t_0 of $[a, b]$. If $\tilde{q}$ is the largest value of q so far calculated then we can reject a subinterval if the Taylor bound on q is less than $\tilde{q}$. One could also obtain bounds on q' to test for monotonicity and even on q'' to test for convexity.

An even simpler rejection criterion follows almost immediately from the following lemma of Stečkin, a proof of which is outlined in [1, Ex. 1.8].

Lemma 3.1 (Stečkin). *Suppose that the polynomial*

$$f(t) = \sum_{|n|\le N} \alpha_n e^{int} \quad (\alpha_n \in \mathbf{C})$$

is real-valued and that $t_0 \in [0, 2\pi]$ satisfies $f(t_0) = \|f\|_\infty$. Then

$$f(t_0 + s) \ge \|f\|_\infty \cos Ns \quad (|s| \le \pi/N). \tag{3.1}$$

Now suppose that q has been evaluated at t_k. If $h < \pi/N$ and a global maximizer t_0 is in the interval $[t_k - h, t_k + h]$ then

$$q(t_k) \ge \|q\|_\infty \cos N(t_k - t_0) \ge \|q\|_\infty \cos Nh \tag{3.2}$$

by Stečkin's lemma. Thus if we know that a global maximizer occurs in one of the intervals $I_1, \dots, I_n$ and we have evaluated q at their midpoints $t_1, \dots, t_n$, then for some k we have $\|q\|_\infty \le q(t_k) \sec Nh$. In particular, if $\tilde{q} := \max\{q(t_i) : i = 1, \dots, n\}$, then the bounds

$$\tilde{q} \le \|q\|_\infty \le \tilde{q} \sec Nh$$

obtain. Moreover if $q(t_i) < \tilde{q} \cos Nh$ then no global maximizer occurs in the interval I_i, which may then be rejected.

These observations give rise to a simple algorithm for calculating $\|p\|_\infty$ described in Figure 3.1.

4. Modified Behaviour near Monomials. As it stands our algorithm exhibits unacceptable behaviour when p is close to (a scalar multiple of) a monomial, for then q is almost constant. Consequently few, if any, of the subintervals are rejected until the

```
Set $h = \pi/M$ for some $M > 2N$.
Subdivide $[0, 2\pi]$ into $M$ subintervals $I_1, \dots, I_n$ of width $2h$.
Repeat
    Calculate $q$ at the midpoints $t_i$ of $I_i$.
    Set $\tilde{q} := \max\{q(t_i) : i = 1, \dots, n\}$.
    If $\tilde{q}(\sec Nh - 1)$ is sufficiently small then Break.
    Reject each $I_i$ with $q(t_i) < \tilde{q} \cos Nh$.
    Subdivide the remaining intervals, reducing $h$ accordingly.
Return $\sqrt{\tilde{q}}$.
```

FIG. 3.1. *The algorithm*

later iterations, resulting in a computational effort increasing exponentially with the accuracy required. However, with a slight modification, we can mollify this behaviour.

Suppose that we have calculated β_n, the coefficients of q:

$$q(t) = \sum_{|n| \le N} \beta_n e^{int}.$$

If $\|q\|_1 - \beta_0 < \epsilon$ then

$$0 \le \|q\|_\infty - \beta_0 \le \|q\|_1 - \beta_0 < \epsilon,$$

the left-hand inequality being just Cauchy's estimate. Thus when p is *very* close to (a scalar multiple of) a monomial we have no work to do — we simply accept $\|p\|_1$ as our estimate of $\|p\|_\infty$.

Next observe that, since q is non-negative, if $A \ge 0$ is a number such that

$$q(t) - A \ge 0 \quad (t \in [0, 2\pi])$$

then $\|q - A\|_\infty + A = \|q\|_\infty$. But we have

$$q(t) = \beta_0 + \sum_{0 < |n| \le N} \beta_n e^{int} \ge \beta_0 - \sum_{0 < |n| \le N} |\beta_n| = 2\beta_0 - \|q\|_1,$$

so $2\beta_0 - \|q\|_1$ will suffice as such a constant A, provided that it is non-negative. Applying Stečkin's lemma to $q - A$ one obtains the improved bounds

$$\tilde{q} \le \|q\|_\infty \le \tilde{q} \sec(Nh) - \big(\sec(Nh) - 1\big) A$$

and the improved rejection criterion that no global maximizer occurs in the interval I_k if

$$q(t_k) < \tilde{q} \cos(Nh) + \big(1 - \cos(Nh)\big) A.$$

From these remarks we obtain a modified algorithm as described in Figure 4.1.

5. Polynomials with Plateaux. The reader may have noticed that the modification to the algorithm will not prevent a large number of polynomial evaluations in the case that the polynomial is almost zero at some point on the circle, while having almost constant absolute value along a significant portion of it. We treat these concerns in some detail in this section.

```
Calculate the coefficients $\beta_n$ of $q$, and $\|q\|_1$.
**If** $\|q\|_1 - \beta_0$ is sufficiently small then **Return** $\|p\|_1$.
Set $A = \max\{0,\, 2\beta_0 - \|q\|_1\}$.
Set $h = \pi/M$ for some $M > 2N$.
Subdivide $[0, 2\pi]$ into $M$ subintervals $I_1, \dots, I_n$ of width $2h$.
**Repeat**
    Calculate $q$ at the midpoints $t_i$ of $I_i$.
    Set $\tilde{q} := \max\{q(t_i) : i = 1, \dots, n\}$.
    **If** $(\tilde{q} - A)\sec(Nh) + A - \tilde{q}$ is sufficiently small then **Break**.
    Reject each $I_i$ with $q(t_i) < (\tilde{q} - A)\cos Nh + A$.
    Subdivide the remaining intervals, reducing $h$ accordingly.
**Return** $\sqrt{\tilde{q}}$.
```

FIG. 4.1. *The modified algorithm*

We consider the periodic function

$$f(t) = \begin{cases} \cos^2(t) & (0 \le t < \pi) \\ 1 & (\pi \le t < 2\pi) \end{cases}$$

which, an elementary calculation shows, has the Fourier series

$$\frac{3}{4} + \frac{1}{4}\cos 2t + \frac{4}{\pi} \sum_{n \ge 1 \text{ is odd}} \frac{1}{n(n^2-4)} \sin nt. \tag{5.1}$$

For odd N we let P_N denote the trigonometric polynomial which is the truncation of this Fourier series at the N-th term, i.e. the series (5.1) but with the summation over $n = 1, 3, \dots, N$. Further let p_N denote the associated order $2N$ polynomial

$$p_N(z) = \left(\frac{3}{4} + \frac{1}{8}(z^2 - z^{-2}) + \frac{2}{i\pi} \sum_{\substack{1 \le n \le N, \\ n \text{ odd}}} \frac{1}{n(n^2-4)}(z^n + z^{-n}) \right) z^N$$

so that $p_N(e^{it}) = P_N(t)e^{iNt}$ and $q_N(t) := \left|p_N(e^{it})\right|^2 = |P_N(t)|^2 = P_N(t)^2$.

LEMMA 5.1. *For each odd $N \ge 1$ the inequalities*

$$\left|f(t)^2 - P_N(t)^2\right| \le 5/\pi N^2 \quad (\pi \le t \le 2\pi), \qquad P_N(\pi/2) \le 2/\pi N^2$$

obtain.

Proof. Since

$$f(t) - P_N(t) = \frac{4}{\pi} \sum_{k=(N+1)/2}^{\infty} \frac{\sin(2k+1)t}{(2k+1)((2k+1)^2-4)}$$

we have, for any t,

$$\begin{aligned} |f(t) - P_N(t)| &\le \frac{4}{\pi} \sum_{k=(N+1)/2}^{\infty} \frac{1}{(2k+1)(4k^2+4k-2)} \\ &\le \frac{1}{2\pi} \sum_{k=(N+1)/2}^{\infty} \frac{1}{k^3} \\ &\le 2/\pi(N+1)^2 \le 2/\pi N^2. \end{aligned}$$

This shows the second inequality, while the observation that

$$\begin{aligned}\left|f(t)^2 - P_N(t)^2\right| &\le |f(t) - P_N(t)|\,|f(t) + P_N(t)|\\ &\le (2/\pi N^2)(2 + 2/\pi N^2)\\ &\le 5/\pi N^2\end{aligned}$$

demonstrates the first. □

Proposition 5.2. *When the modified algorithm is applied to the polynomial p_N ($N \ge 3$, N odd) then no intervals in $[\pi, 2\pi]$ are rejected while*

$$h > \frac{1}{N^2}\sqrt{\frac{10}{\pi}}.$$

Proof. Using the notation above, the rejection criteria of the modified algorithm is that

$$q_N(t_i) < \tilde{q}_N \cos 2Nh + (1 - \cos 2Nh)A \tag{5.2}$$

since p_N is of order $2N$. However, by Lemma 5.1,

$$\begin{aligned}&\tilde{q}_N \cos 2Nh + (1 - \cos 2Nh)A\\ &\qquad\le \left(1 + 5/\pi N^2\right)\cos 2Nh + (1 - \cos 2Nh)\left(2/\pi N^2\right)^2\\ &\qquad\le \left(1 + 5/\pi N^2\right)(1 - 4N^2h^2/3) + 8h^2/\pi^2 N^2 \qquad (2Nh \le \pi/2)\end{aligned}$$

since $1 - x^2/3 \ge \cos x \ge 1 - x^2/2$ for $|x| \le \pi/2$. Hence

$$\begin{aligned}&\tilde{q}_N \cos 2Nh + (1 - \cos 2Nh)A\\ &\qquad\le 1 + 5/\pi N^2 + (8/\pi^2 N^2 - 4N^2/3)h^2\\ &\qquad\le q_N(t_i) + 10/\pi N^2 + (8/\pi^2 N^2 - 4N^2/3)h^2 \qquad (2Nh \le \pi/2)\end{aligned}$$

by another application of the lemma. Now, for $N \ge 2$, we have

$$8/\pi^2 N^2 - 4N^2/3 \le -N^2$$

so the inequality (5.2) cannot obtain if

$$10/\pi N^2 - N^2 h^2 < 0 \qquad (N \ge 3,\ h \le \pi/4N).$$

Finally we note that the initial conditions on h given by the algorithm guarantee that $h \le \pi/4N$, so this last inequality is that which we sought to prove. □

We can now establish a lower bound on the number of polynomial evaluations required by the modified algorithm when p_N is its input. Since the relative error ϵ of the algorithm's output is $\sec 2Nh - 1$, a short calculation shows that $h \le (3\epsilon)^{1/2}/2N$. Thus with a stopping condition specifying that the relative error is less than $40/3\pi N^2$ we have a final value of h no greater than $(10/\pi)^{1/2}/N^2$. If, at each subdivision stage, we have divided each remaining interval into m subintervals then we can be sure that, at some point in the algorithm's execution, h satisfies

$$\frac{1}{N^2}\sqrt{\frac{10}{\pi}} < h \le \frac{m}{N^2}\sqrt{\frac{10}{\pi}}.$$

Then, by the proposition, no subintervals in $[\pi,\, 2\pi]$ have been rejected for this value of h and so at least

$$\frac{\pi}{2h} \geq \frac{N^2\pi}{2m}\sqrt{\frac{\pi}{10}} \approx \frac{0.88N^2}{m} \tag{5.3}$$

polynomial evaluations are required.

In the following table we detail the actual number of polynomial evaluations required for a few small values of N. The figures were obtained from the author's implementation of the algorithm which uses a triadic subdivision (i.e. $m = 3$).

N	ϵ	final h	evaluations	
			lower bound	actual
5	0.169765	0.041391	8	59
15	0.018863	0.004896	67	411
35	0.003465	0.002138	360	949
105	0.000385	0.000240	3236	7379

The estimate of (5.3) is something of a disappointment since it shows that the modified algorithm has, to all intents and purposes, worst case performance no better than $O(N^2)$ polynomial evaluations, i.e. $O(N^3)$ elementary operations if Horner's rule is used for the evaluation of the polynomial.

6. Evaluation. In spite of the results of the last section on worst-case performance the modified algorithm *seems* to be much more efficient in the average case. We emphasize that our evidence for this claim is entirely empirical.

The author's implementation[1] of the modified algorithm was evaluated on two counts. Firstly to evaluate the distribution of required evaluations. The histogram in Figure 6.1 depicts the distribution of the number of evaluations required to calculate the maximum modulus of an order 20 polynomial to an accuracy of 10^{-10}. The sample taken was of of 20,000 randomly chosen polynomials (i.e. the coefficients of the polynomials were chosen to have their real and imaginary parts uniformly distributed over $[-1, 1]$). In this sample all of the polynomials required evaluations in the range 87 to 255, contrasting with the 1079 required to evaluate the order 18 polynomial p_9 to the same accuracy.

Secondly the average required evaluations are displayed in Figure 6.2. There we plot the average number of evaluations required to calculate the maximum modulus of polynomials to an accuracy of 10^{-10} against their order (the orders being chosen to be approximately exponentially distributed). The average was taken over 100 randomly chosen (in the same sense as above) complex polynomials of each order.

In each case the calculations required for these figures took around 20 minutes to run on a Sun SPARCstation.

7. Conclusions. The algorithm that we have discussed in this note has the advantages that is simple to describe and implement, and seems to offer good average performance. However it has poor worst-case performance, even if empirical evidence suggests that such behaviour is rare. In this respect the algorithm has similarities with well-known techniques for evaluating the eigenvalues of matrices (though we do not suggest that our results are so deep).

[1]The author's implementation of the algorithm for RLAB, Ian Searle's linear algebra package, is available on request.

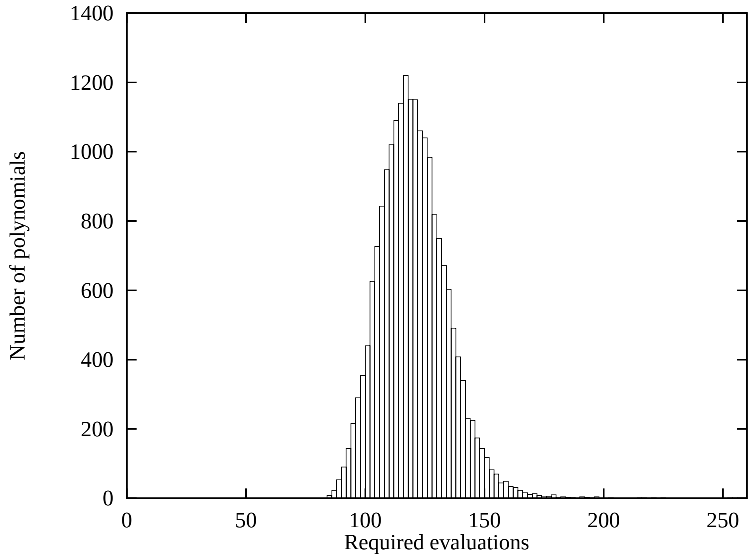

FIG. 6.1. *Distribution of required evaluations.*

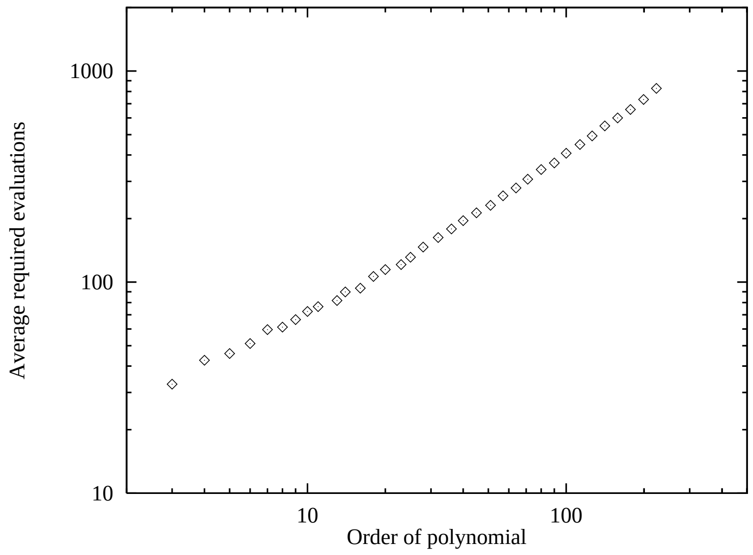

FIG. 6.2. *Required evaluations.*

To obtain an algorithm with better worst-case performance one could consider a hybrid approach whereby the modified algorithm is applied until a predetermined number (say $10N$) polynomial evaluations have been performed, and then proceed using a different method, possibly one based on bounds obtained from Taylor's theorem.

Finally we mention one possible direction for further improvement of the algorithm. We observe that non-rejected subintervals occur in blocks (around the local maximizers of q) and so evaluation of q at the midpoints of these subintervals could be performed extremely efficiently using the generalized chirp transform as described in [5, Prop. 1.4].

REFERENCES

[1] R. E. Edwards, *Fourier Series 1*, vol. 64 of Graduate Texts in Mathematics, Springer-Verlag, New York, second ed., 1979.
[2] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, *C++ Toolbox for Verified Computing 1*, Springer-Verlag, Berlin, 1995.
[3] E. Hansen, *Global optimization using interval analysis — the one-dimensional case*, J. Optim. Theory Appl., 29 (1979), pp. 247–270.
[4] D. J. Kavvadias and M. N. Vrahatis, *Locating and computing all the simple roots and extrema of a function*, SIAM J. Sci. Comput., 17 (1996), pp. 1232 – 1248.
[5] J. H. Reif, *Approximate complex polynomial evaluation in near constant work per point*, in 29th Annual ACM Symposium on Theory of Computing, El Paso, Texas, 1996. A PostScript version of this paper can be obtained from `http://www.cs.duke.edu/~rief/paper/Eval.ps`.
[6] B. Sz.-Nagy, *Unitary Dilations of Hilbert Space Operators and Related Topics*, no. 19 in CBMS Regional Conference Series, AMS, Providence, RI, 1974.

Appendix A. Rlab Implementation[2]**.** For rapid prototyping the algorithm is implemented in Rlab, Ian Searle's high-level matrix computation language[3]. A translation of this routine for Matlab would be straight-forward. We create a file `maxmod.r`

"maxmod.r" 8a $\equiv$
⟨ Statement of syntax 8b ⟩
⟨ Declarations 9a ⟩
⟨ The function `maxmod()` 9b ⟩
◇

A.1. Preliminaries. The file will need a short description of its purpose since the Rlab help command will echo the file.

⟨ Statement of syntax 8b ⟩ $\equiv$

```
// Synopsis:  Calculate the maximum modulus of a complex
//    polynomial on the unit disc.

// Syntax:  <<p;t;h;evals>> = maxmod ( p , options )

// Description:

// The maximum modulus of a complex polynomial p on the unit
// disc is estimated using a method of repeated subdivision
// of [0,2pi] and rejection of non-maximizing subintervals.
// The rejection criterion is based on a lemma of Steckin.

// options is a list with (possibly) the members
//   relerr precision required [default 0.5e-8]
//   verbose   set non-zero for run-time information
//   intervals    initial number of intervals [default 5*p.n]

// maxmod() returns a list with the members
//   p   estimate of the maximum modulus of p
//   t   vector of midpoints of non-rejected subintervals
//        h half-width of each interval
```

[2]The implementation description is not intended for publication.
[3]Rlab can be obtained via `www.eskimo.com/~ians/rlab`

```
//   evals  number of evaluations of the polynomial

// This rfile is generated from maxmod.w, the nuweb source
// file, which contains a full description of the algorithm.
//      maxmod.w can be obtained from the author J.J Green at
//      jjg1@cise.npl.co.uk
```
◇

Macro referenced in 8a.

The RLAB functions that we need but which are not built-in must be specified by a `require` statement. We also specify supporting functions and variables to be `static` (in the C sense).

⟨ Declarations 9a ⟩ ≡

```
// declarations
require conv
static(TRUE,FALSE);
TRUE=1; FALSE=0;
```
◇

Macro referenced in 8a.

A.2. The `maxmod()` function. The RLAB function that calculates $\|p\|_\infty$. The first argument of `maxmod()` is a vector specifying the coefficients. The n-th element of the vector is the coefficient of z^{n-1} in the polynomial. The second (optional) argument is a list containing the options (see Section A.3).

There is a slight complication in calculating the stopping criterion for the algorithm since we wish to terminate with a user defined relative accuracy of ϵ in the value of our estimate of $\|p\|_\infty$, whereas our calculations are in terms of $\|q\|_\infty$. Rather than calculate computationally expensive square-roots we take a different approach.

For the moment write Q_U and Q_L respectively for some upper and lower bounds on the value of $\|q\|_\infty$, Q for their mean and P for $\sqrt{Q}$. If for some $\delta > 0$ we have $Q_U - Q_L < \delta$ then

$$\|q\|_\infty - Q = (\|p\|_\infty - P)(\|p\|_\infty + P)$$

so that

$$|\,\|p\|_\infty - P| = \frac{|\,\|q\|_\infty - Q|}{\|p\|_\infty + P} \leq \frac{\delta}{2(\|p\|_\infty + P)}.$$

Thus the relative error of P as an estimate of $\|p\|_\infty$ is certainly dominated by $\delta/4Q_L$. It follows that to obtain a required accuracy of ϵ in the algorithm it suffices to insist that $Q_U - Q_L < 4\epsilon Q_L$, the stopping criterion that appears in the code fragment below.

⟨ The function `maxmod()` 9b ⟩ ≡

```
maxmod=function(p,options){
  global(pi);
  ⟨ Get options 12b ⟩
  ⟨ Calculate coefficients $\beta_n$ of $q$ 10a ⟩
  if (normq-beta0 < 4.*epsilon.*beta0){
    ⟨ Very close to a monomial 10b ⟩
    else {
      totalEvals=0;upperBound=inf();lowerBound=0;
      ⟨ Calculate $A$ 10c ⟩
      ⟨ Create initial interval-set 11a ⟩
      ⟨ Print statistics header 12c ⟩
```

```
          while (1) {
            ⟨ Evaluate upper and lower bounds 11c ⟩
            if (upperBound-lowerBound<4.*epsilon.*lowerBound){
              ⟨ Print final statistics 13 ⟩
              break;
              }
            ⟨ Reject non-maximizing subintervals 11d ⟩
            ⟨ Print statistics 12d ⟩
            ⟨ Subdivide remaining intervals 12a ⟩
            ⟨ Calculate q values 11b ⟩
            }
          }
        }
      output.p=sqrt((lowerBound+upperBound)/2);
      if (exist(interval)){
        output.t=interval[;1];
        output.h=h;
        }
      output.evals=totalEvals;
      return output;
      };
    ◇
```

Macro referenced in 8a.

⟨ Calculate coefficients β_n of q 10a ⟩ ≡

```
    // The use of conv() below would be better replaced by a direct
    // calculation (conv() uses fft with the attendant overhead)
    beta=conv(p[p.n:1:-1;1],conj(p));
    beta0=real(beta[p.n]);
    normq=sum(abs(beta));
    ◇
```

Macro referenced in 9b.

⟨ Very close to a monomial 10b ⟩ ≡

```
    lowerBound=beta0;
    upperBound=normq;
    totalEvals=0;
    ◇
```

Macro referenced in 9b.

⟨ Calculate A 10c ⟩ ≡

```
    A=max([2.*beta0-normq,0]);
    ◇
```

Macro referenced in 9b.

Details on the intervals are stored in the matrix `interval[]`. To reduce storage we have all intervals of width $2h$ and store only the midpoint t and the value of $q(e^{it})$. Thus `interval[]` is a 2-column real matrix.

The initial interval set consists of evenly-spaced points on the unit disc. Hence we can evaluate the values of q extremely quickly using the fast fourier transform.

⟨ Create initial interval-set 11a ⟩ ≡

```
// initial interval-set
h=pi/options.intervals;
interval=zeros(options.intervals,2);
interval[;1]=(2.*h).*[0:options.intervals-1]';
pvalues=fft(p,options.intervals);
interval[;2]=real(pvalues).^2+imag(pvalues).^2;
evalCount=options.intervals;
totalEvals=options.intervals;
```

◇
Macro referenced in 9b.

Calculation of the values of q is the inner-loop for this program and efforts to improve performance should be concentrated here. A common way of getting better performance in Rlab (as with Matlab) is to *vectorize* the code, i.e. replace for-loops with operations on vectors. In the following we implement a vectorized Horner's rule for this calculation.

⟨ Calculate q values 11b ⟩ ≡

```
// calculate q-values
ind=find(interval[;2]==0); // the q-values not known
evalCount=ind.n;           // for statistics
totalEvals=totalEvals+evalCount;
zvalues=exp(interval[ind;1].*1i); // new z-values
pvalues=p[ones(1,ind.n)];
for (k in 2:p.n){
  pvalues=pvalues.*zvalues+p[k.*ones(1,ind.n)];
  } // p(z) calculated using vectorized Horner's rule
interval[ind;2]=real(pvalues).^2+imag(pvalues).^2;
```

◇
Macro referenced in 9b.

⟨ Evaluate upper and lower bounds 11c ⟩ ≡

```
// evaluate bounds
lowerBound=max(interval[;2]);
SteckinConstant=cos(N.*h);
upperBound=(lowerBound-A)./SteckinConstant+A;
```

◇
Macro referenced in 9b.

⟨ Reject non-maximizing subintervals 11d ⟩ ≡

```
// rejection
rejectionThreshold = (lowerBound-A).*SteckinConstant+A;
keepind=find(interval[;2]>rejectionThreshold);
rejectCount=interval.nr-keepind.n;
interval=interval[keepind;];
```

◇
Macro referenced in 9b.

We divide each interval into three (rather than two) since then we already know the values of q at the midpoint of the middle subinterval. Of course any odd number would do, but we aim not to sample in detail where it is not needed.

⟨ Subdivide remaining intervals 12a ⟩ ≡

```
// subdivision
h=h/3;
offset=2.*h;
tvalues=interval[;1];
interval=[interval;zeros(2.*interval.nr,2)]; // preallocate
interval[tvalues.nr+1:interval.nr;1]=...
  [tvalues+offset;tvalues-offset];
```

◇
Macro referenced in 9b.

A.3. Supporting Routines.

⟨ Get options 12b ⟩ ≡

```
// get options
if (!exist(p)){error("First argument must be a vector!");}
if (p.nc>1){p=p[:];}
N=p.nr-1;
if (!exist(options)){options=<<>>;}
if (!exist(options.relerr)){
  epsilon=0.5e-8;
  else {
    epsilon=options.relerr;
    }
  }
if (epsilon==0){error("Can't achieve this level of accuracy!");}
if (!exist(options.verbose)){options.verbose=FALSE;}
verbose=(options.verbose!=FALSE);
if (!exist(options.intervals)){options.intervals=ceil(5.*N);}
if (options.intervals < 2.*N){
  printf("Steckin method requires opt.intervals > 2N\n");
  error();
  }
```

◇
Macro referenced in 9b.

⟨ Print statistics header 12c ⟩ ≡

```
if (verbose){
 printf("lower bound\tupper bound\tevals\trejected\n");
 }
```

◇
Macro referenced in 9b.

⟨ Print statistics 12d ⟩ ≡

```
if (verbose){
  printf("%f\t%f\t%i\t%i\n",...
   sqrt(lowerBound),sqrt(upperBound),evalCount,rejectCount);
  }
```

◇
Macro referenced in 9b.

⟨ Print final statistics 13 ⟩ ≡

```
        if (verbose){
          printf("%f\t%f\t%i\t-\n",...
           sqrt(lowerBound),sqrt(upperBound),evalCount);
          }
```

◇

Macro referenced in 9b.

A.4. Index of Macros.

A.5. Index of Variables.