Augmenting a Programming Language with Complex Arithmetic

W. Kahan
Univ. of Calif. @ Berkeley

J. W. Thomas
Apple Computer, Inc.

Abstract: Currently, programming languages that support COMPLEX arithmetic as well as REAL oblige compilers to implement certain obscure optimizations lest expressions mixing REAL with COMPLEX variables engender superfluous or even harmful computation. This can be avoided if the language provides, for the compiler's use, a third IMAGINARY data-type about which the programmer need know only the name of the language's imaginary unit i or I or j. And on computers conforming to IEEE 754/854, the scheme proposed here honors complex conjugation in ways that other schemes cannot.

Current Practice

A COMPLEX number **Z** is usually rendered as a pair **Z** = (X, Y) of REAL numbers X and Y. Doing so is unwise because confusing an object with its representation usually spawns subtle nuisances. This practice leads to extra work when COMPLEX expressions mix with REAL; a REAL R has to be coerced to COMPLEX (R, 0) before it can participate in complex arithmetic. Besides wasting time upon manipulations of 0, this practice sometimes corrupts results unnecessarily. For example, if R*(X, Y) must first be coerced to (R, 0)*(X, Y) before it yields (R*X-0*Y, R*Y+0*X) instead of simply (R*X, R*Y), then 3.0*(∞, 5.0) will yield not (∞, 15.0) but (∞, NaN) in arithmetic conforming to IEEE standards 754/854 for floating-point arithmetic. Similarly a sum R + (X, Y) that should yield (R+X, Y) gets coerced instead to (R+X, 0+Y), which changes the sign of Y if it is -0.0 and thus spoils computations of conformal maps of slitted domains. We shall see two examples of maps spoiled this way at the end.

To some extent these nuisances can be ameliorated by precluding premature coercions; the compiler can be instructed to consult a table of special formulas for mixtures of REAL with COMPLEX expressions. This expedient does not do justice to pure imaginary expressions Yi that must be represented as pairs (0, Y) with that annoying zero. For example, no way exists for $(\infty i)^2$ to be computed as -∞ instead of (-∞, NaN) without wasteful tests at run-time for zero operands in complex arithmetic operations. To do better, we have to recognize IMAGINARY as a data-type.

One Pure IMAGINARY Constant

Consider a programming language with one REAL data-type but not yet encumbered with a COMPLEX data-type. Our task now is to add COMPLEX to the syntax of the language in a way that matches it most closely to the desired traditional mathematical semantics, even if we have to build a structure more elaborate than some programming languages have used in the past. A measure of our success will be the ease with which a mathematician can read and understand a program written to do exactly what he or she intends.

We start with the declaration

IMAGINARY ɿ

which names the imaginary unit ɿ that satisfies ɿ*ɿ = ɿ² = −1 . The language's arbiters of taste and fashion can choose any other name, like *i* or *j* or *%i* , so long as it is the same for all users of that language; this accord lets programs share COMPLEX data in ASCII files without first finding out what name the data used for ɿ . Programs that perform COMPLEX arithmetic must use this declaration to inform the compiler that the name for ɿ is henceforth not to be used to name an integer nor anything else but the imaginary unit. Without this declaration, ɿ could stand for a variable, or ɿ(...) a function. The best way to avoid errors of this kind is to use ɿ only as a suffix, language permitting.

A programmer need never declare anything else to be IMAGINARY .

Variables and functions may be declared REAL or COMPLEX in the usual way. Then the language classifies *expressions* as REAL, IMAGINARY or COMPLEX according to the following rules:

REAL expressions and constants are recognized as usual. Also, products or quotients of even numbers of IMAGINARY expressions are REAL, as are values of certain functions like ABS(...) , REAL(...) and IMAG(...) that can take arguments of all three types. The product/quotient rule will be explained further later.

Anything of the form ɿ*(REAL Expression) or (REAL Expression)*ɿ or ɿ(REAL Expression) or (REAL Expression)ɿ (this is best) is IMAGINARY ; moreover, REAL literal constants may be either followed by ɿ as a suffix or multiplied by ɿ to become of type IMAGINARY, as are the examples ɿ , 1ɿ , 1*ɿ , ɿ*1 , 1.0E0ɿ . And products or quotients of (perhaps no) REAL expressions with odd numbers of IMAGINARY expressions are IMAGINARY. Like REAL numbers, IMAGINARY numbers occupy one floating-point register or one memory cell of adequate width. Although an expression like ɿ*(REAL Expression) looks formally like a product, it is never actually multiplied but merely promoted to (REAL Expression)ɿ or ɿ(REAL Expression) of type IMAGINARY. It's like a *cast* in C .

The order of formal multiplication by ɿ should not matter; all four expressions ɿ*Y , Y*ɿ , ɿ(Y) and (Y)ɿ amount to the same thing, but may get there by different routes at compile time.

COMPLEX = REAL + IMAGINARY

Every COMPLEX expression is a formal sum of one REAL and one IMAGINARY expression stored in adjacent cells. There is no need to write X + ɿ*Y as (X, Y) , so the latter is left available for the language to use as a tuple or list. In other words, any expression of type (REAL) + (IMAGINARY) is not actually added but merely promoted to type COMPLEX. And the rules for mixing COMPLEX with COMPLEX, REAL or IMAGINARY expressions are expressed in that form using a decomposition like **Z** = Z_R + (Z_I)ɿ for every COMPLEX variable or expression **Z** . (Accidently either Z_R or Z_I could vanish, leaving the COMPLEX variable **Z** with a pure imaginary or real value respectively but not turned into an expression of IMAGINARY or REAL type; neither does the REAL X := 3.0 inherit the type INTEGER from its integer value.)

Apparently, all complex rational arithmetic except division by a COMPLEX number has to be built into the language's grammar; the rest can be handled via procedure-calls, possibly in-line.

In what follows, R, U, X are REAL expressions or variables; ℇS, ℇV, ℇY are abbreviations for IMAGINARY expressions ℇ(S) etc. derived from products ℇ*S etc.; and bold **T**, **W**, **Z** are COMPLEX expressions with **T** = R + ℇS , **W** = U + ℇV , **Z** = X + ℇY formally. CONJ(X + ℇY) := X + ℇ(−Y) is the *complex conjugate* operation; all rational operations must be so implemented that they *commute* with it despite roundoff, as for instance does multiplication: CONJ(**W*****Z**) = CONJ(**W**)*CONJ(**Z**) . CONJ(X) = X for REAL X and CONJ(ℇY) = ℇ(−Y) for IMAGINARY ℇY , of course.

Here are tables that describe a grammar for complex evaluation:

Multiply	X	ℇY	Z = X + ℇY
U	R := U*X	ℇS := ℇ(U*Y)	**T** := (U*X) + ℇ(U*Y)
ℇV	ℇS := ℇ(V*X)	R := −V*Y	**T** := (−V*Y) + ℇ(V*X)
W = U + ℇV	**T** := U*X + ℇ(V*X)	**T** := −V*Y + ℇ(U*Y)	**T** := (U*X − V*Y) + ℇ(U*Y + V*X)

Add	X	ℇY	Z = X + ℇY
U	R := U+X	**T** := U + ℇY	**T** := U+X + ℇY
ℇV	**T** := X + ℇV	ℇS := ℇ(V+Y)	**T** := X + ℇ(V+Y)
W = U + ℇV	**T** := U+X + ℇV	**T** := U + ℇ(V+Y)	**T** := U+X + ℇ(V+Y)

Subtraction is similar.

Divide	X	ℇY	Z = X + ℇY
U	R := X/U	ℇS := ℇ(Y/U)	**T** := (X/U) + ℇ(Y/U)
ℇV	ℇS := ℇ(−X/V)	R := Y/V	**T** := (Y/V) + ℇ(−X/V)
W = U + ℇV	**T** := X/**W**	**T** := ℇY/**W**	**T** := **Z**/**W**
	Division by a COMPLEX number calls appropriate subroutines that are described below.		

Assignments and Arguments of Functions

An assignment like " **Z** := ℇ(Y) " must create a COMPLEX value for **Z** := 0 + ℇY just as " **Z** := X " must create **Z** := X + 0ℇ . Programers who wish to avoid those extra zeros can do the same for pure IMAGINARY variables as would be done for REAL ; retain just that REAL value and postpone its binding into a COMPLEX context for as long as possible. This strategy is most valuable for arguments of functions that cost much more to evaluate for arbitrary COMPLEX arguments than for REAL or pure IMAGINARY.

For example,

$$\exp(\imath Y) = \cos(Y) + \sin(Y)\,\imath\ ,$$

$$\sin(\imath Y) = \sinh(Y)\,\imath \quad \text{and} \quad \sinh(\imath Y) = \sin(Y)\,\imath\ ,$$

$$\cos(\imath Y) = \cosh(Y) \quad \text{and} \quad \cosh(\imath Y) = \cos(Y)\ ,$$

$$\tan(\imath Y) = \tanh(Y)\,\imath \quad \text{and} \quad \tanh(\imath Y) = \tan(Y)\,\imath\ ,$$

$$\arcsin(\imath Y) = \operatorname{arcsinh}(Y)\,\imath \quad \text{and} \quad \operatorname{arcsinh}(\imath Y) = \arcsin(Y)*\imath\ ,$$

$$\arctan(\imath Y) = \operatorname{arctanh}(Y)*\imath \quad \text{and} \quad \operatorname{arctanh}(\imath Y) = \arctan(Y)\,\imath\ .$$

(No similarly simple identity relates arccos and arccosh .)

A compiler for a language that, like Fortran, possesses GENERIC INTRINSIC functions could ideally expedite this strategy of delay by invoking substitutions like those above automatically whenever it recognized IMAGINARY arguments. This cannot always be easy, so programmers should not demand too much help from the compiler. No compiler can resolve ambiguities that arise when functions are discontinuous across *slits*, as are arcsin(Y) and arctanh(Y) for $Y^2 > 1$, and $\sqrt{x}$ for $x < 0$. Consequently, identities above for arcsinh($\imath$Y) and arctan($\imath$Y) yield IMAGINARY values only if Y is a constant and $-1 \le Y \le 1$. And only for constant $x < 0$ may compilers substitute $\sqrt{x} = \sqrt{(-x)}\,\imath$ safely; $\sqrt{x}$ should signal *Invalid* at run-time for a REAL variable $x < 0$.

Some Subtleties

A number of subtleties complicate complex arithmetic. Some come from the plethora of infinities that can descend from division by zero or from deserved overflows in practically any operation; a fair treatment of these infinities must be deferred to some other occasion. Other subtleties concern the avoidance of undeserved and harmful over/underflows that render unsatisfactory for complex division $(X + \imath Y)/(U + \imath V)$, say, a traditional formula like $((X*U + Y*V) + \imath(Y*U - X*V)) / (U^2 + V^2)$ because its denominator can too easily over/underflow prematurely.

The procedures presented below will avoid most undeserved over/underflows and also avoid the worst consequences of roundoff, a third subtlety. A fourth concerns the sign of zero; this will be treated well enough below to justify the attention paid above to the preservation of zero's sign despite tradition to the contrary.

Only in complex square root will explicit attention be paid to the sign of zero; elsewhere its correct handling is implicit in the rules for arithmetic conforming to IEEE 754/854, or impossible in other arithmetics. Consequently, CONJ commutes with SQRT only for IEEE 754/854, not for other arithmetics. The same is so for inverse functions like ln, arctan, arctanh, arcsin, ..., but they will not be discussed further here. The point here is that the IMAGINARY type has been introduced not merely to save a few arithmetic operations with zeros but more to help ensure that every complex operation shall conform to all applicable standards. Later, two examples will show how far awry simple calculations can go if zero's sign is not handled as IEEE 754/854 stipulates.

Some Subroutines

Robert Smith's algorithm for computing

$$R + \imath S := (X + \imath Y)/(U + \imath V)$$

when $U + \imath V \neq 0 + \imath 0$ goes as follows:

```
If  |U| > |V|  then
                P := V/U ;                  Simpler versions
                Q := U + V*P ;              of this algorithm
                R := (X + Y*P)/Q ;          must be derived
                S := (Y - X*P)/Q            and used in those
            else                            cases when it is
                P := U/V ;                  known at compile-
                Q := U*P + V ;              time that  X = 0
                R := (X*P + Y)/Q ;          or that  Y = 0 ;
                S := (Y*P - X)/Q            see table above.
            end if.
```

On machines with relatively slow division, another algorithm that scales the denominator $U + \imath V$ before using the unsatisfactory but simpler formula given earlier, then scales the quotient, can run faster and just as reliably.

To compute $R := ABS(X + \imath Y) := \sqrt{(X^2 + Y^2)}$ without severe damage from premature over/underflow:

```
If  |X| < |Y|  then  swap( X, Y )  end if ;
If  X = 0  then  R := 0  else  R := √((Y/X)^2 + 1)*|X|  end if.
```

Ideally complex square root should never overflow, but perfection is more costly than is needed for our examples. Instead we tender an algorithm valid if $(X + \imath Y)*(1+\sqrt{2})^{\pm 1}$ would not over/underflow. To compute $R + \imath S := SQRT(X + \imath Y) := \sqrt{(X + \imath Y)}$:

```
R := √( (ABS(X + ıY) + |X|)/2 ) ; ... over/underflow hurts here.
If  R = 0  then  S := Y
           else  if  X > 0  then  S := (Y/R)/2
                            else
                                 S := CopySign(R, Y) ;
                                 R := (Y/S)/2
                            end ifs.
```

The function CopySign is specified for arithmetics that conform to IEEE 754/854 ; CopySign(R, Y) has the same magnitude as R but the same sign bit as Y even if Y is ± 0 . This implements the square root's discontinuity along the negative real axis in such a way as ensures that CONJ(SQRT(Z)) = SQRT(CONJ(Z)) for all COMPLEX Z including $Z < 0$. For example CONJ(−4 + 0ı) = −4 − 0ı and SQRT(−4 ± 0ı) = 0 ± 2ı respectively. (No such identity can hold if the computer's arithmetic lacks −0 or mishandles it; in that case CopySign(R, 0) might as well agree with Fortran's SIGN(R, 0.0) := +|R| as if 0.0 had a " + " sign by convention, though that will roil conformal maps like the two examples below.)

The algorithms above are not accurate enough to produce Gaussian integers exactly, but they are accurate enough for our examples.

Apropos of algorithmic accuracy, it is worth noting briefly that Z^2 and, for integers $N > 2$, higher powers Z^N often computed by repeated squaring, are usually obtained more accurately from

$$(X + \imath Y)^2 := (X-Y)*(X+Y) + \imath(X*Y + X*Y)$$

in two multiplications and three add/subtractions than from

$$(X + \imath Y)^2 := X^2 - Y^2 + \imath(X*Y + X*Y)$$

in three multiplications and two add/subtractions.

Two Examples: Eluding Flow past a Disk, and Borda's Mouthpiece
Let $f(Z) := (Z - 1/Z)/2$ and $g(W) := W - i\sqrt{(iW-1)}\sqrt{(iW+1)}$. Do not " simplify " $g(W)$ to $W - i\sqrt{(-W^2-1)}$ nor to $W - \sqrt{(W^2+1)}$ since they behave differently. Though $f(g(W)) = W$ for all W , $g(f(Z)) = Z$ only for all $|Z| \geq 1$ and some $|Z| = 1$; otherwise $g(f(Z)) = -1/Z$. Deducing where these identities hold is tricky.

As a conformal map, $W = f(Z)$ maps the complex Z-plane twice, once for $|Z| \geq 1$ and once for $|Z| \leq 1$, onto the complex W-plane, mapping the unit circle $|Z| = 1$ to a slit along the imaginary axis from $W = -i$ to $W = +i$. The inverse map is the one we wish to plot; $Z = g(W)$ maps the whole W-plane slitted along $W = -i$ to $W = +i$ onto the outside of the Z-plane's unit circle $|Z| \geq 1$. Vertical lines in the W-plane map to the stream-lines of a vertical "eluding" flow around the unit circle in the Z-plane. We wish to exhibit those stream-lines.

To smooth the plot near stagnation points $g(\pm i)$ where $g'(\pm i)$ is infinite, we parameterize the vertical W-lines in a way that plots points more densely near the stagnation points than far away from them; the real function $h(s) := (3s^5 - 10s^3 + 15s)/8$ does this by satisfying $h(-s) = -h(s)$, $h'(s) \geq 0$, $h(1) = 1$ and $h'(1) = 0$. Then for any fixed real r , $W := r + ih(s)$ traces out a vertical straight line segment as s runs through some interval, say $-1.5 \leq s \leq 1.5$, in small steps like $\Delta s = 3/32$; and $Z = g(W)$ traces out a stream-line past the circle.

To plot several stream-lines, we run r through some interval, say $0 \leq r \leq 0.6$, in several steps, say of size $\Delta r = 0.05$. For each such r we plot two stream-lines, $Z := g(r + ih(s))$ to the right of the circle and $Z := g(-r + ih(s))$ to the left. The kind of result we expect is shown in Figure 1 , and that is what does happen if -0 is respected. But machines that lack -0 or spoil it plot two coincident stream-lines around the right-hand arc of the circle and none around the left, as Figure 2 shows.

Of course, some fiddling with tiny perturbations (but not too tiny lest they get lost in rounding errors) can bring back the complete circle; but why should that be necessary? As problems get more complicated, the effects of omitting -0 get more bizarre. Try plotting $Z := 1 + W^2 + W\sqrt{(W^2+1)} + \ln(W^2 + W\sqrt{(W^2+1)})$ as W runs on radial straight lines through 0 in the right half-plane, including the imaginary axis. The flow, called "Borda's Mouthpiece", should look like Figure 3 ; but Figure 4 shows what happens without -0 nor fiddling. Can you explain it?

For more details about the phenomena in question, and for more carefully coded procedures to compute the above complex inverse elementary functions, see " Branch Cuts for Complex Elementary Functions, or Much Ado About the Sign of Zero " by W. Kahan, ch. 7 in *The State of the Art in Numerical Analysis* ed. by Iserles & Powell (1987), Oxford Univ. Press. Updated versions of this document are released from time to time by its author.

Acknowledgments The first author's work has been supported for many years by the U. S. Office of Naval Research (N00014-90-J-1372) and recently also by the National Science Foundation (CCR- 8812843 and ASC-9005933).

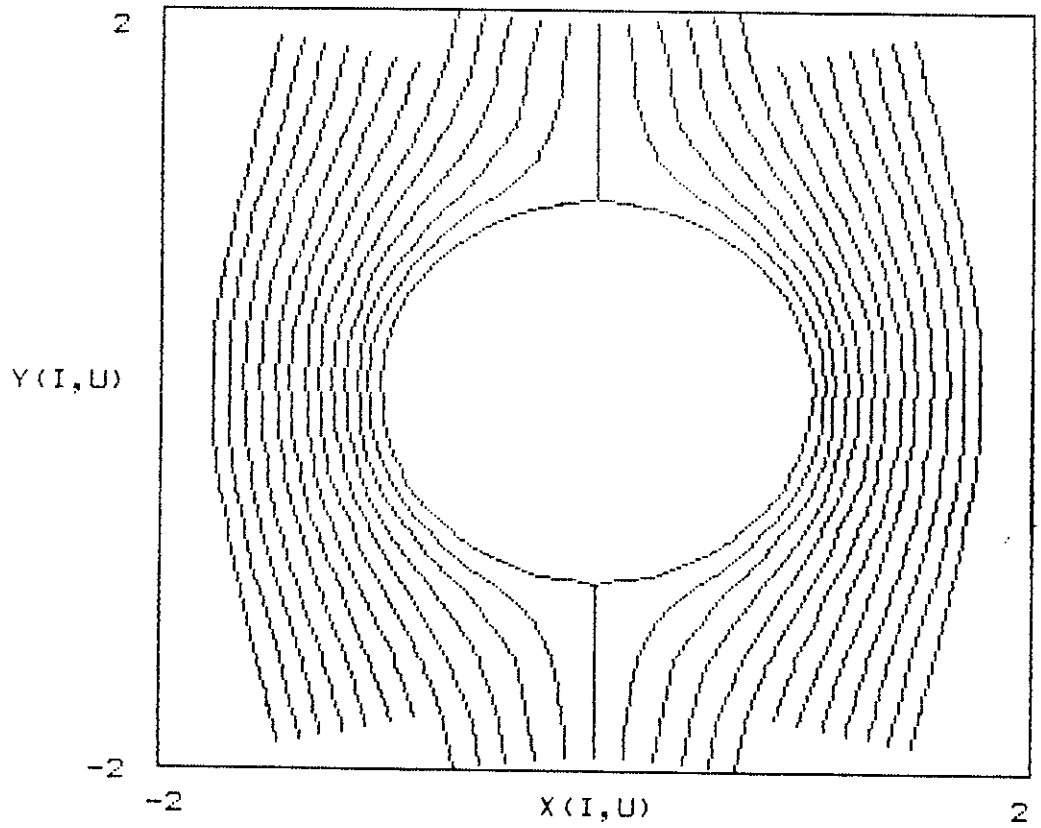

Figure 1 : Eluding Flow Past the Unit Disk

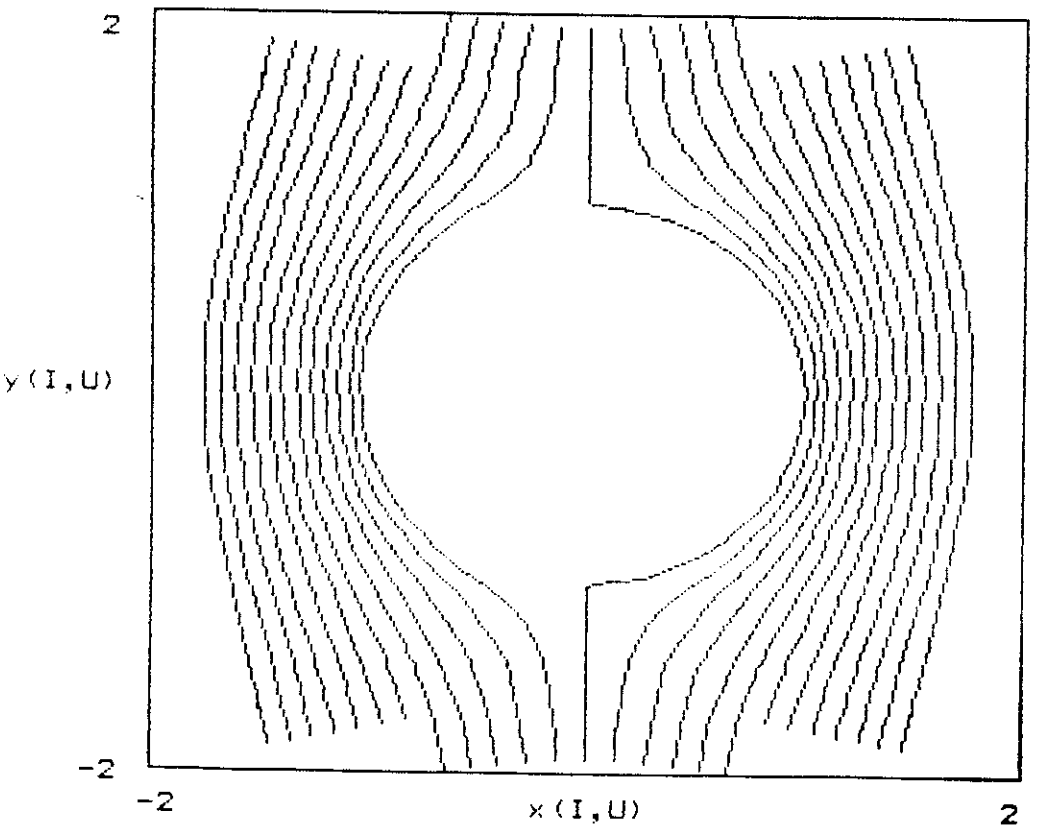

Figure 2 : Eluding Flow Past the Unit Disk, Almost

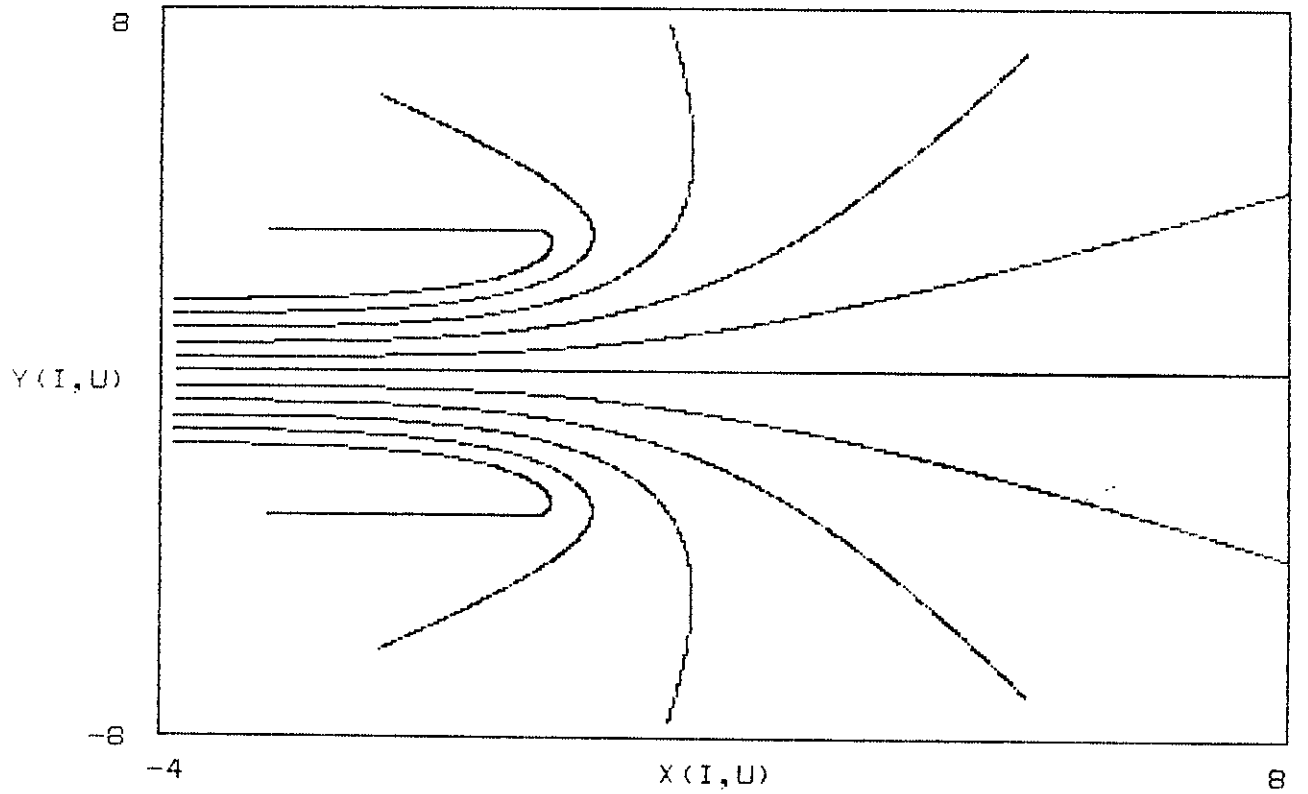

Figure 3 : Borda's Mouthpiece

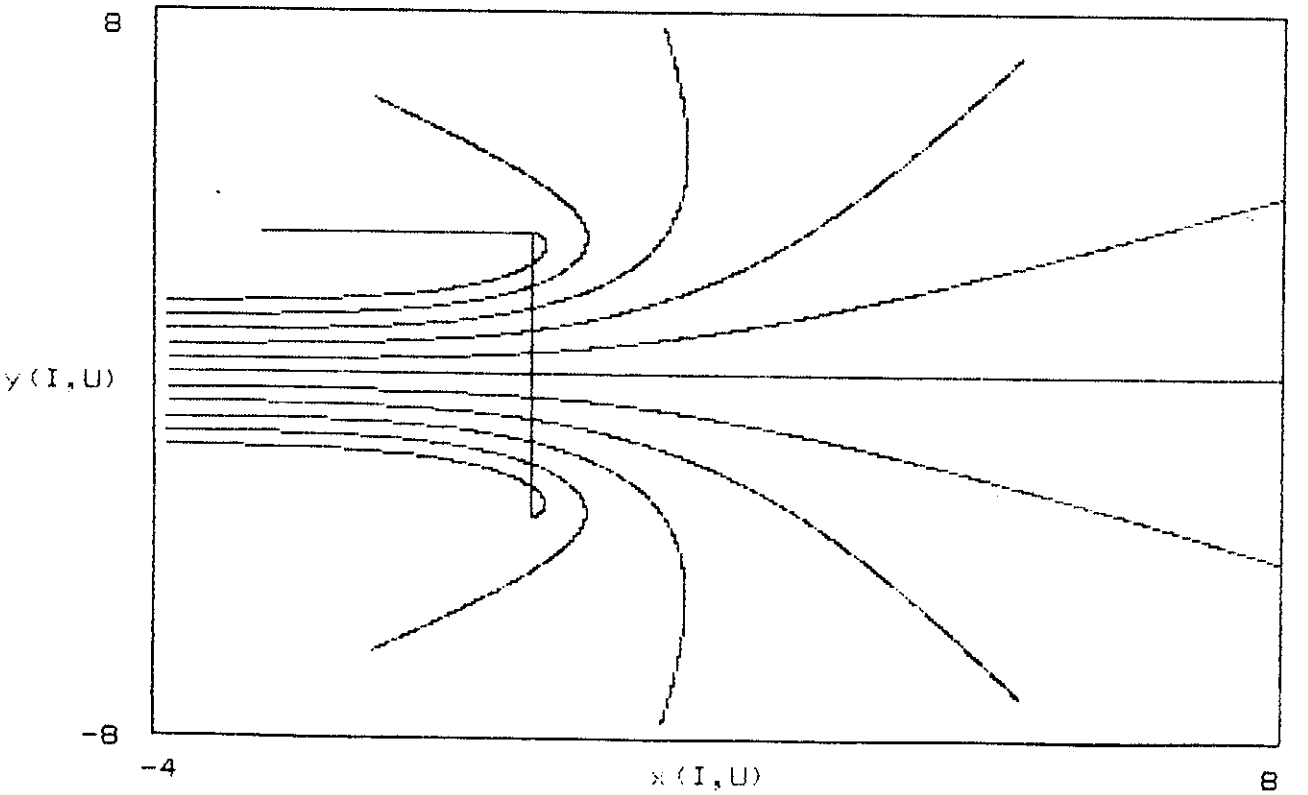

Figure 4 : Borda's Mouthpiece, Almost