# 5
# Procedimentos Computacionais

Neste capítulo descreveremos as principais rotinas do **wx2x2**, utilizadas extensivamente no processo de inversão.

A partir do momento em que o programa possui os valores dos parâmetros globais, assim como as expressões das funções coordenadas $F_1$ e $F_2$, a rotina *study_function()* efetua a chamada de várias outras sub-rotinas que, em linhas gerais, realizam as seguintes tarefas: calcular o grau da função, localizar e construir as primeiras curvas críticas e computar sua imagem, coletar a informação necessária para que se proceda aos testes de contagem e à geração de palavras de Blank-Troyer.

À medida que o conjunto crítico $\mathcal{C}$ vai sendo calculado, o programa decompõe curvas críticas e suas imagens em *arcos bimonotônicos* (ao longo dos quais as duas coordenadas são monótonas) e, a partir dessa decomposição, calcula os números de rotação e giração de várias curvas. Cúspides são localizadas com precisão, junto com sua classificação como interna ou externa. O programa calcula também as intersecções de curvas em $F(\mathcal{C})$, permitindo nomear as placas no domínio e na imagem. A seguir, são gerados grafos de adjacência entre placas, necessários para os testes de contagem e Blank-Troyer, empregados para a identificação exaustiva do conjunto crítico $\mathcal{C}$.

Uma vez que essa estrutura está disponível, o programa pode desenhar a flor da função ou inverter pontos, processos que, por sua vez, exigem uma quantidade de outras sub-rotinas. Para começar, são gerados quatro pontos na imagem para os quais é possível nesse momento calcular todas as pré-imagens. A partir desses pontos resolvidos iniciais, outros podem ser invertidos por métodos de continuação ao longo de caminhos $\beta$ especiais. A inversão da função ao longo de $\beta$ é natural em situações em que os arcos de $F^{-1}(\beta)$ não encontram pontos críticos. Nos pontos críticos é necessário proceder com mais cuidado: ao trespassar uma dobra, por exemplo, o número de pré-imagens de pontos em $\beta$ aumenta ou diminui de 2. É conveniente também garantir que os caminhos $\beta$ a inverter evitem pontos associados a uma análise numérica inutilmente difícil, como ocorre quando $\beta$ passa próximo à imagem de uma cúspide. O programa tem estratégias para selecionar caminhos de inversão e

ainda acumula informação em um banco de pontos resolvidos, isto é, listas de pontos com suas pré-imagens já calculadas, selecionados por um conjunto de critérios.

Nas próximas seções, são descritas com mais detalhe várias das sub-rotinas acima.

## 5.1
## Calculando o grau

A rotina *find_degree()* calcula o grau de $F$, baseada no Teorema 3.1. Como $F$ é supostamente cordata, seu grau é o número de rotação $\omega(F \circ \gamma, 0)$, onde $\gamma$ é uma circunferência centrada na origem, orientada positivamente, de raio suficientemente grande. Amostramos $\gamma$ em $N$ pontos $p_i$ distribuídos uniformemente. Uma variável inteira *four_degree* é atualizada quando $F(p_i)$ e $F(p_{i+1})$ estão em quadrantes consecutivos: ela aumenta ou diminui de 1, dependendo da orientação. Ao completar a volta, dividimos *four_degree* por 4 e, assim, obtemos o grau da função $F$.

## 5.2
## Detectando e construindo curvas críticas

Inicialmente percorremos uma malha quadrangular cuja densidade e amplitude são especificadas através dos parâmetros globais *NGRID*, *Grid step* e *Grid ratio*, em busca das primeiras curvas críticas, que inicializam o conjunto $\mathcal{C}_\bullet$. Para cada uma dessas curvas, determinamos seu sentido de dobra, calculamos suas cúspides e as classificamos em interna ou externa. Inicializamos o conjunto $F(\mathcal{C}_\bullet)$ que contém a imagem de cada curva em $\mathcal{C}_\bullet$, computamos seus pontos de intersecção e, para cada curva em $F(\mathcal{C}_\bullet)$, calculamos sua giração. A seguir, o par $\mathcal{C}_\bullet$-$F(\mathcal{C}_\bullet)$ é submetido a três tipos de testes, que buscam estabelecer se $\mathcal{C}_\bullet = \mathcal{C}$. Cada teste é efetuado em cada placa de $\mathcal{C}_\bullet$, e se algum desses testes falha em determinada placa, é porque existe ao menos uma curva crítica que não está presente em $\mathcal{C}_\bullet$. Então, na placa onde ocorreu a falha, o programa efetuará uma busca para encontrar uma nova curva crítica, que é acrescentada ao conjunto $\mathcal{C}_\bullet$.

Quando todos os testes dão certo, o **wx2x2** entende que $\mathcal{C}_\bullet = \mathcal{C}$. Os resultados teóricos apenas caracterizam os conjuntos críticos $\mathcal{C}$ e suas imagens $F(\mathcal{C})$ de funções cordatas. Isso não garante que o conjunto $\mathcal{C}_\bullet$ e sua imagem $F(\mathcal{C}_\bullet)$ são de fato o conjunto crítico completo da função dada $F$: eles podem ser o conjunto crítico de uma outra função $\hat{F}$. O problema, em um certo sentido, é insolúvel: curvas críticas podem ser adicionadas em qualquer região arbitrariamente pequena de um ponto regular de uma função $F$, sem alterar

a função fora desta região. Qualquer método numérico, que só se permita calcular uma função em um número finito de pontos, não pode alcançar a certeza de ter encontrado todas as curvas críticas. Os testes, entretanto, são muito expressivos: é difícil esconder curvas críticas deles. Ainda assim, existem parâmetros globais que aumentam o refinamento da busca por curvas críticas, entre os quais *Curve Sharpness*, *Maximum Step*, *Minimum Step*.

## 5.3 Detectando as primeiras curvas críticas

A seguir, descrevemos como uma curva $\Gamma$ é representada no programa. Os atributos da classe CriticalCurve são:

1. *index*: inteiro que rotula $\Gamma$, começando por 0;
2. *domBoundingBox*: coordenadas extremas em $x$ e $y$ dos pontos de $\Gamma$;
3. *imgBoundingBox*: coordenadas extremas em $x$ e $y$ dos pontos de $F(\Gamma)$;
4. *orientation*: informa o sentido de dobra em $\Gamma$;
5. *domPoints* e *imgPoints*: pontos da discretização de $\Gamma$ e suas imagens;
6. *numPoints*: o total de pontos da discretização em $\Gamma$;
7. *cusps*: dá a localização das cúspides, internas e externas;
8. *numInCusps* e *numOutCusps*: contam cúspides internas e externas;
9. *intersections*: lista os pontos de intersecção das curvas na imagem;
10. *domCriticalsX* e *domCriticalsY*, *imgCriticalsX* e *imgCriticalsY*: listam os extremos locais das coordenadas $x$ e $y$ em $\Gamma$ e $F(\Gamma)$;
11. *turningNumber*: a giração de $F(\Gamma)$;
12. *winding(q)*: uma rotina que calcula o número de rotação de $\Gamma$ em torno do ponto $q$ fornecido como entrada;

Todas as curvas críticas calculadas pelo programa, junto com suas imagens, são guardadas num vetor que está disponível a qualquer momento.

Agora, passamos a descrever o processo de busca por curvas críticas. No início da rotina *study_function()*, percorremos uma malha quadrangular centrada na origem cuja geometria foi especificada através dos parâmetros globais *NGRID*, *Grid step* e *Grid ratio*. Procuram-se pontos consecutivos $p_i$ e $p_{i+1}$ nos quais os sinais de $\det DF$ nestes pontos sejam diferentes, o que indica

a existência de um ponto crítico no segmento $\overline{p_i p_{i+1}}$. Se o programa ainda não registrou (pela rotina *test_stick()*) a existência de um ponto crítico em $\overline{p_i p_{i+1}}$, a rotina *find_cr_point()* calcula ali um ponto crítico novo $r$, que serve de ponto de partida para *make_cr_curve()*, que por sua vez realiza a construção da curva crítica por $r$.

A rotina *make_cr_curve()* cria um objeto da classe CriticalCurve e usa *next_zero()* várias vezes, para gerar seqüencialmente, por argumentos de análise local, pontos críticos que são adicionados ao vetor *domPoints* do objeto. Assim constrói uma curva crítica que será orientada pelo sentido de dobra, quando completada. As rotinas *next_zero()* e *find_cr_point()*, juntas, implementam um método de continuação numérica, do tipo preditor-corretor, onde a etapa preditora é de passo variável. A variação do passo é necessária para levar em conta o fato freqüente de que na mesma função existem curvas críticas de tamanhos e afastamentos muito diferentes. O usuário pode interferir nessa etapa ajustando os parâmetros globais *Curve Sharpness*, *Minimum step*, *Maximum step* e *Min step in make_cr_curve()*, existentes na opção *Preferences* do **wx2x2**.

Essencialmente, *make_cr_curve()* realiza as seguintes tarefas:

1. Prepara as condições iniciais para invocar *next_zero()*.

2. Determina o término do cálculo da curva crítica (à medida que *next_zero()* vai calculando pontos críticos supostamente novos, é necessário garantir que os novos pontos não sejam discretizações de um arco da curva já percorrido).

3. Determina o sentido de dobra da curva, estabelecendo se a curva crítica foi construída no sentido anti-horário ou horário. Note que essa informação é global: ela está disponível só quando a curva está completa. A orientação é obtida examinando a vizinhança de um ponto cuja coordenada $x$ é máxima.

Para o funcionamento de *next_zero()*, são necessárias algumas construções geométricas. O vetor tangente $r_t(r)$ à uma curva crítica num ponto $r = (x_0, y_0)$ é obtido girando o vetor gradiente $\nabla g(p)$ de 90° em sentido horário, onde $g(x, y) = \det(DF(x, y))$, para que a curva assim construída fique orientada em sentido de dobra. Como $g(x_0, y_0) = 0$, $\nabla g(r)$ é aproximado por

$$\begin{aligned} \nabla g(r) &= (g_x(r), g_y(r)) \\ &\approx \left( \frac{g(x_0 + \Delta, y_0)}{\Delta}, \frac{g(x_0, y_0 + \Delta)}{\Delta} \right), \ \Delta > 0 \text{ pequeno.} \end{aligned}$$

Daí,

$$r_t(r) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \nabla g(r) = \left( \frac{g(x_0, y_0 + \Delta)}{\Delta}, -\frac{g(x_0 + \Delta, y_0)}{\Delta} \right)$$

A rotina *next_zero()* recebe dois pontos consecutivos de uma curva crítica e, a partir deles, calcula um novo ponto. Para realizar a primeira chamada à *next_zero()*, como *make_cr_curve* dispõe de apenas um ponto crítico $r$ usamos o vetor tangente $r_t(r)$ para calcular um quase ponto crítico $r_{-1} = r - step \cdot r_t(p)$ e, assim realizamos a primeira chamada à rotina *next_zero()* passando os pontos $r_{-1}$ e $r$. O valor *step* na definição de $r_{-1}$ é calculado a partir de uma análise local em $r$ para levar em consideração a curvatura da curva crítica em $r$: quanto maior a curvatura, menor será o valor de *step* (seu valor deve ser maior ou igual a *Min step in make_cr_curve()*).

Agora explicaremos como, na rotina *next_zero()*, obtemos um novo ponto crítico a partir de outros dois consecutivos. Sua implementação se baseia em duas etapas (Figura 5.1):

**Etapa Preditora:** Seja $\gamma$ uma curva crítica de $F$ contendo dois pontos $r_0$, $r_1$ calculados seqüencialmente. Calculamos $q = r_1 + sec$ a partir do vetor secante $sec = 2 \cdot step \cdot \frac{r_1 - r_0}{|r_1 - r_0|}$ e os pontos $q_a = q + R(-90) \cdot sec$ e $q_b = q + R(+90) \cdot sec$, onde $R(\theta)$ representa a matriz de rotação por $\theta$ graus. Se os sinais de $\det DF$ em $q_a$ e $q_b$ forem distintos, refinamos o vetor *sec* levando em conta o parâmetro global *Curve Sharpness*, para fins de robustez, e depois executamos a Etapa Corretora. Caso contrário, o vetor *sec* sofre uma contração de fator 0.5 e os pontos $q$, $q_a$ e $q_b$ são atualizados. Se num número de pré-determinando de vezes, não obtivermos sucesso em encontrar $q_a$ e $q_b$ com sinais distintos, a construção da curva crítica passando por $r$ é abortada.

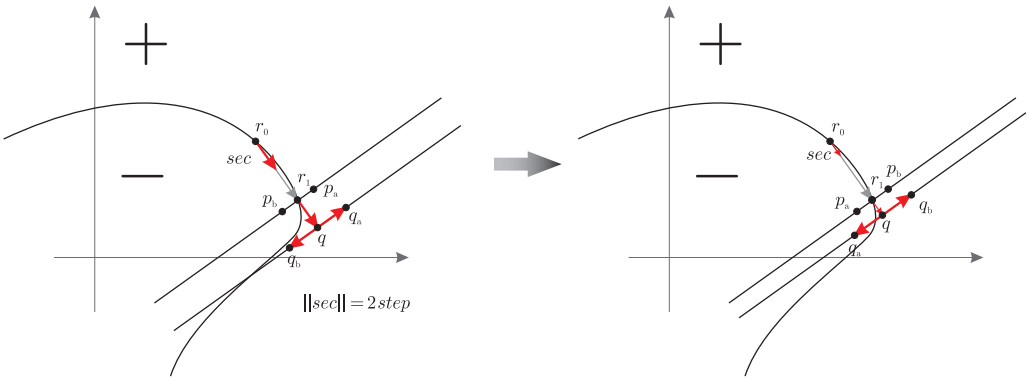



Figura 5.1: Aqui é necessário reduzir *step*

**Etapa Corretora:** a rotina *find_cr_point()* é chamada para calcular um ponto crítico no segmento $\overline{q_a q_b}$

A rotina *find_cr_point()*, crucial no funcionamento de *next_zero()*, recebe dois pontos $m$ e $n$ nos quais os sinais de $\det DF$ são distintos e retorna um ponto crítico no segmento $\overline{mn}$. O algoritmo de localização de zeros empregado é o *método de Brent* (ou *Van Wijngaarden-Dekker-Brent*), que combina os métodos da bisseção, da secante e interpolação quadrática inversa. É um método robusto, no sentido que desempenha pelo menos tão bem quanto bisseção e ainda assim pode ser tão rápido quanto métodos mais instáveis. Na maioria das iterações, o algoritmo emprega os métodos da secante ou interpolação quadrática inversa para aumentar a velocidade de convergência. O método de Brent é recomendado na bibliografia para calcular zeros de uma função real de variável real quando apenas seus valores estão disponíveis, e não suas derivadas ([PR]).

Em *next_zero()*, na Etapa Corretora, a rotina *find_cr_point()* é chamada recebendo como parâmetros $m$ e $n$ os pontos $q_a$ e $q$ ou $q_b$ e $q$, respectivamente se $\det(DF(q_a)) \cdot \det(DF(q)) < 0$ ou $\det(DF(q_b)) \cdot \det(DF(q)) < 0$. E, assim, o ponto crítico procurado é da forma $r = (1 - t_0) \cdot m + t_0 \cdot n$, onde $t_0$ um zero para a função $\det(DF((1 - t) \cdot m + t \cdot n))$ em $[0, 1]$.

Ao término da construção da nova curva crítica marcamos os segmentos da malha que foram atravessados por essa nova curva (isso é feito para evitar repetições na busca por outras curvas) e retomamos a travessia da malha. Ao terminar de percorrer toda a malha original, algumas curvas críticas são encontradas, formando o conjunto $\mathcal{C}_\bullet$, representado por um vetor de objetos do tipo CriticalCurve.

### 5.3.1
### Calculando alguns atributos de CriticalCurve

Explicamos aqui como alimentamos alguns atributos da classe CriticalCurve. À medida que uma curva crítica nova é construída, verifica-se as propriedades extremais em relação às coordenadas $x$ e $y$ tanto da curva quanto dos pontos de sua imagem: assim são gerados *domCriticalsX*, *domCriticalsY*, *imgCriticalsX* e *imgCriticalsY*, assim como *domBoundingBox* e *imgBoundingBox*, dimensões dos retângulos mínimos que contém as curvas. Com os pontos extremos determinados, temos em mãos a decomposição da curva crítica e sua imagem em arcos bimonotônicos, que consistem de arcos de pontos entre dois pontos extremos consecutivos. (Figura 5.2).

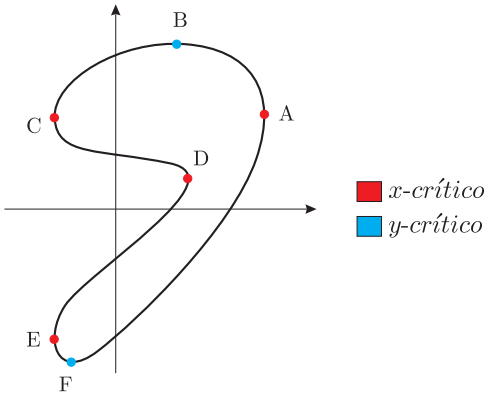



Figura 5.2: Curva dividida em arcos bimonotônicos

Identificamos também pontos candidatos a vizinhos de cúspides, isto é, pontos $p_1$, $p_2$ e $p_3$ no domínio para os quais há uma súbita mudança de direção entre os vetores $F(p_1) - F(p_2)$ e $F(p_2) - F(p_3)$ (Figura 5.3). Vizinhos bem calculados a uma cúspide são necessários para a classificação interna-externa, depois que o sentido de dobra é calculado. A rotina *refine_cusp()* refina o trecho de arco crítico determinado por $p_1$, $p_2$ e $p_3$ para que a rotina *seek_cusp()* localize e classifique uma cúspide nesse arco (caso exista) e, logo após, os atributos *cusps*, *numInCusps* e *numOutCusps* da curva crítica são atualizados.

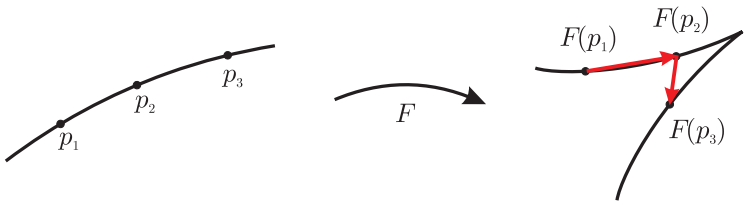
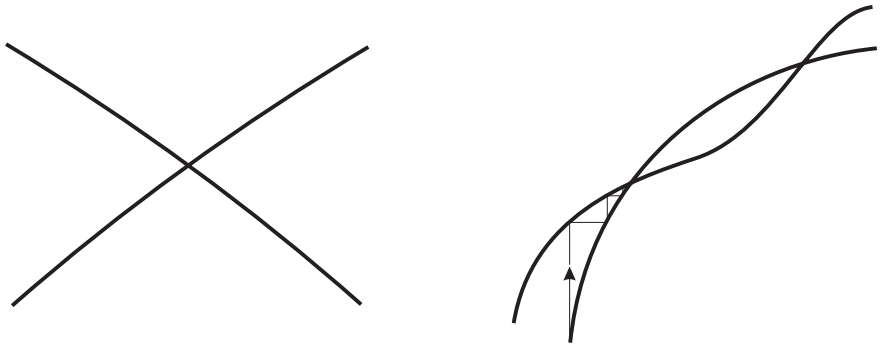



Figura 5.3: Localizando cúspides

Procuramos também por pontos de intersecção no conjunto $F(\mathcal{C}_\bullet)$, uma primeira utilização da decomposição das curvas em arcos bimonotônicos. Para cada curva em $F(\mathcal{C}_\bullet)$ procuramos por pontos de auto-intersecção e pontos de intersecção com outras curvas. Para identificar nas listas de pontos do domínio e da imagem os vizinhos de pontos de intersecção procedemos da seguinte maneira. Dadas duas imagens de curvas críticas, verificamos inicialmente se seus *bounding boxes* têm intersecção não vazia. Neste caso, passamos a considerar todos os pares de arcos bimonotônicos, um em cada curva. Mais uma vez, se os bounding boxes desses arcos não se interceptam, não há nada a fazer.



(a) (b)

Figura 5.4: Intersecções entre arcos bimonotônicos

Caso contrário, existem dois sub-casos que são ilustrados na Figura 5.4. No caso 5.4a, uma pesquisa binária nas listas de pontos dos arcos encontra o ponto de intersecção (se existir). No caso 5.4b, os pontos de intersecção são encontrados pelo algoritmo sugerido na Figura. Na realidade, esses dois métodos fornecem apenas vizinhos ao ponto de intersecção e, para obter precisamente a intersecção, inserida nas curvas tanto do domínio quanto na imagem, procedemos a uma análise numérica local.



### 5.3.2 Girações e rotações

Aqui explicaremos como calcular números de rotação, girações e relações de inclusão entre as curvas críticas de $\mathcal{C}_\bullet$. Essa informação é necessária para a implementação da rotina *belongsToPlaque()*, uma função que decide em que placa (do domínio ou da imagem) está um ponto.

Descrevemos o cálculo do número de rotação de uma curva $\gamma$ em torno da origem (uma translação reduz o caso geral a esse). Na rotina *winding()*, percorremos os pontos extremos dos arcos bimonotônicos da curva e verificamos mudanças de quadrante entre pontos extremos consecutivos. Mudanças entre quadrantes vizinhos somam ou subtraem um quarto de volta ao número de rotação de acordo se a mudança ocorreu no sentido anti-horário ou horário, respectivamente. Se dois pontos extremos de um arco bimonotônico estão em quadrantes opostos, soma-se ou subtrai-se meia volta, mas agora é necessário invocar a rotina *x_to_pt()* para determinar se a origem está abaixo ou acima do arco. Na figura abaixo, as frações indicam as contribuições de cada arco bimonotônico para o cálculo do número de rotação da curva em torno à origem.

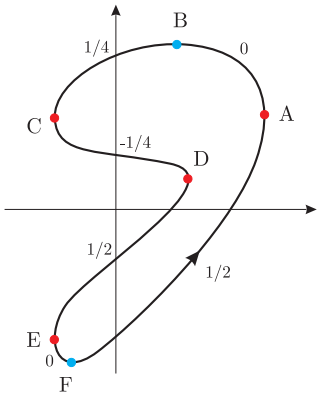



Figura 5.5: Uma volta em torno à origem

Para calcular girações de curvas do contradomínio também utilizamos a decomposição da curva em arcos bimonotônicos. Em pontos extremos suaves, o vetor tangente claramente pertence a um dos quatro raios determinados pelos eixos coordenados. Quando os arcos se encontram na imagem de uma cúspide somamos meia volta à giração. Assim podemos inicializar o atributo *turningNumber* (a giração) para cada curva em $F(\mathcal{C}_\bullet)$.

### 5.3.3
### $\mathcal{C}_\bullet = \mathcal{C}$?

Três tipos de testes — de *coerência*, de *contagem* e de *Blank-Troyer* — são aplicados ao conjunto $\mathcal{C}_\bullet$ para determinar se este conjunto está completo, isto é, se todas as curvas críticas foram localizadas. Esses testes são realizados em cada placa de $\mathcal{C}_\bullet$: quando temos uma reprovação em alguma placa, o programa volta a procurar outras curvas críticas nessa placa.

O teste de coerência consiste em verificar se duas curvas vizinhas por inclusão possuem orientações contrárias: afinal, percorrer a curva no sentido de dobra faz com que a região à direita tenha jacobiana com determinante positivo. Isso é feito comparando o atributo *orientation* das curvas durante uma travessia do grafo de adjacência.

Os testes de contagem são as verificações do Teorema 3.7. Para cada placa $X$ do domínio, todos os ingredientes necessários para aplicar esse teste foram calculados nas seções anteriores, a saber: grau da função, girações e o número de cúspides internas e externas.

O teste de Blank-Troyer é implementado conforme descrito em [MST1], e baseia-se na caracterização de conjuntos críticos e suas imagens. Esse teste pode ser habilitado ou desabilitado definindo o parâmetro global *Blank*, na opção *Preferences* do **wx2x2**, conforme descrito na seção 4.4.

### 5.3.4
### Procurando outras curvas críticas

Quando uma placa $X$ de $\mathcal{C}_\bullet$ não passa num dos testes mencionados acima é porque existe alguma curva crítica ainda não detectada no interior de $X$. Passamos então a procurar por dois pontos $p_0$ e $p_1$ em $X$ nos quais $\det DF$ tem sinais opostos. A partir deste segmento, a rotina *find_cr_point()* calcula um ponto crítico, que é estudado pela rotina *test_san()* para determinar sua pertinência a alguma curva crítica já calculada. Se o ponto é considerado novo, *make_cr_curve()* constrói a nova curva.

Uma reprovação no teste de coerência é tratada da forma descrita a seguir. Suponhamos $X$ limitada. Curvas na fronteira de $X$ são classificadas como *boas* ou *más*. A fronteira exterior é boa. Fronteiras interiores são boas se sua orientação dada pelo sentido de dobra é oposta à da fronteira exterior de $X$; as demais curvas são más. A reprovação de $X$ no teste de coerência significa que existe pelo menos uma curva má. Entre as curvas más, seja $p'$ o ponto com coordenada $y$ mínima. Dentre os pontos *em curvas boas* com coordenada $y$ menor ou igual à de $p'$, tomamos $p''$ minimizando a distância a $p'$ na métrica $d((x_0, y_0), (x_1, y_1)) = \max\{|x_0 - x_1|, |y_0 - y_1|\}$, obtido como em (I) abaixo. Obtenha agora no segmento $\overline{p'p''}$ dois pontos interiores $p_0$ e $p_1$ vizinhos aos vértices, usados agora como parâmetros para *find_cr_point()*: o ponto crítico encontrado será aprovado por *test_san()*, a menos de algum problema numérico.

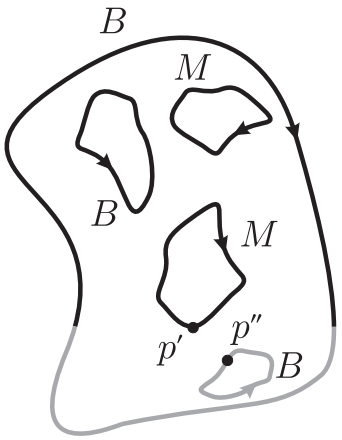



Figura 5.6: Encontrando uma curva crítica nova

Na métrica definida acima, o ponto mais próximo a $p$ num arco bimonotônico $\widehat{\mathrm{AB}}$ é $A$, $B$ ou uma intersecção de $\widehat{\mathrm{AB}}$ com uma das duas retas passando por $p$ com inclinação $\pm 1$. Um simples teste verificando a localização do *bounding box* do arco nos diz em qual caso no encontramos. Para procu-

rar a interseção com uma reta, uma pesquisa binária entre os pontos do arco desempenha bem.

Suponhamos agora que a placa $X$ foi aprovada no teste de coerência, mas reprovada num dos outros dois testes. Em $X$, o determinante da jacobiana em pontos próximos à fronteira têm um sinal bem definido, digamos positivo. Estamos procurando por pontos em $X$ com determinante da jacobiana negativa, a partir do qual *find_cr_point()* gera um novo ponto candidato a inicializar *make_cr_curve()*.

Para começar, geramos aleatoriamente um ponto $p$ em $X$. Isso se faz escolhendo pontos dentro do *bounding box* da fronteira exterior e verificando sua pertinência a $X$ com a rotina *belongsToPlaque()*. Se $p \in X$ satisfaz $\det(DF(p)) < 0$, basta invocar *find_cr_point()*. Se isso não acontece, procuramos diminuir o determinante ao longo de uma reta por $p$ alinhada com uma discretização de $\text{grad}(\det(DF))(p)$. A busca se repete ao longo de segmentos sempre em $X$, por um certo número de vezes. Se o ponto desejado não é encontrado, outro ponto $p$ é sorteado. O processo é cotado pelo parâmetro global *Number of tries*.

## 5.4 Invertendo um ponto

Agora estamos prontos para descrever o processo de resolução numérica da equação

$$F(x) = q, \qquad x,\, q \in \mathbb{R}^2. \tag{5-1}$$

Uma das tarefas principais é a inversão de $F$ por um *método de continuação* ao longo de um segmento $\overline{q_0 q_1}$ na imagem para o qual uma pré-imagem $p_0$ de $q_0$ é conhecida. O segmento é discretizado e novas pré-imagens são obtidas por um método de Euler-Newton a partir das pré-imagens anteriores ([AG]). No **wx2x2**, a rotina *cont_pre_im()* implementa esse método, tendo como parâmetros de entrada os pontos $q_0$, $q_1$ e $p_0$. A rotina retorna os pontos $p_1$, $p_*$ e um indicador que informa um dos três resultados a seguir:

a) a rotina foi bem-sucedida e a pré-imagem desejada é $p_1$;

b) a rotina descobriu que o processo de continuação não vai além de um ponto $p_*$. Isto ocorre quando o segmento $\overline{q_0 q_1}$ intercepta $F(\mathcal{C})$, sendo $q_* = F(p_*)$ a intersecção;

c) ocorrência de um erro numérico na rotina, indicada na área de log.

A resolução de (5-1) é realizada em três etapas:

a) cálculo de todas as pré-imagens de alguns pontos;

b) obtenção de um caminho $\beta$, em forma de "L" (veja Figura 5.8), composto por dois segmentos de reta, $\beta_0$ e $\beta_1$, satisfazendo certos critérios descritos abaixo;

c) inversão dos segmentos $\beta_0$ e $\beta_1$.

### 5.4.1
### Calculando todas as pré-imagens de alguns pontos

A rotina *init_invert()* é responsável por esta tarefa. Inicialmente procuramos um *bounding box* $B_{\mathcal{C}}$ para as curvas de $\mathcal{C}$ e outro $B_{F(\mathcal{C})}$ para as curvas de $F(\mathcal{C})$. Tomamos a seguir um ponto $p$ fora de $B_{\mathcal{C}}$ com imagem $q = F(p)$ fora de $B_{F(\mathcal{C})}$. Isto é possível porque $F$ é cordata. Em outras palavras, $q$ está na placa ilimitada de $F(\mathcal{C})$. A partir de $q$, completamos um quadrado como na Figura 5.7).

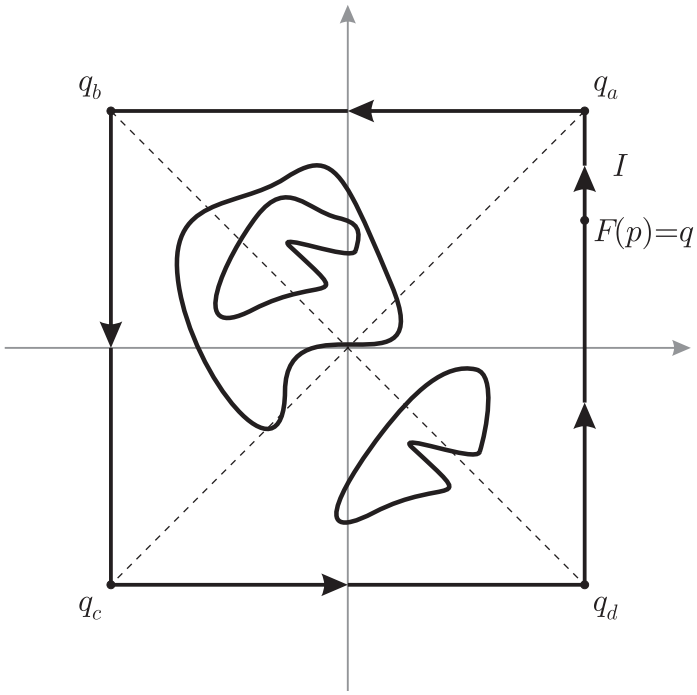



Figura 5.7:

Obtemos uma pré-imagem $p_a$ de $q_a$, através da rotina *cont_pre_im()*, usada invertendo por continuação ao longo do segmento $I$, com a condição inicial $p$.

Depois, usando $p_a$ como uma condição inicial para inverter ao longo de $\overline{q_a q_b}$, obtemos uma pré-imagem de $q_b$. Invertendo os outros lados do quadrado, obtemos pré-imagens de $q_c$ e $q_d$ e uma outra pré-imagem $p'_a$ de $q_a$. Repetimos então o ciclo de inversões ao longo dos lados do quadrado a partir de $p'_a$, até obtermos o número total de pré-imagens de cada um desses quatro pontos,

que deve ser, segundo o Corolário 3.4, igual a $|\deg(F)|$. Aliás, $\deg(F)$ já foi calculado pela rotina *find_degree()*. Se este processo falhar por razões numéricas, começamos novamente selecionado um ponto $p$ mais longe da origem. No programa, a cada falha aplicamos uma escala de 1.2 ao ponto $p$ anterior, ou seja, $p \leftarrow 1.2 \cdot p$. Pelo Teorema 3.5, este processo deve funcionar para $p$ suficientemente longe da origem.

Em conclusão, temos quatro *pontos resolvidos* $q_a$, $q_b$, $q_c$ e $q_d$, cujas pré-imagens são todas conhecidas. Estes pontos e suas pré-imagens são armazenados no *banco de pontos resolvidos* e jamais são removidos. O banco é útil para a execução de inversões de pontos genéricos: dado $q_1$ na imagem, selecionamos um ponto $q_0$ no banco de pontos resolvidos, ligamos $q_0$ a $q_1$ por um caminho em forma de "L" e invertemos, por continuação, de $q_0$ até $q_1$, algumas pré-imagens de $q_1$ a partir das pré-imagens de $q_0$.

Para exemplificar, vamos considerar novamente a função $\widetilde{F}(z) = z^2 + \bar{z}$. A inversão ao longo dos segmentos $\beta_0$ e $\beta_1$ saindo de $q_0$ a partir das pré-imagens $p_0^0$ e $p_0^1$ dão origem a duas pré-imagens de $q_1$, $p_1^0$ e $p_1^1$. É claro, entretanto, que duas pré-imagens estão faltando, como deixa claro a figura. Essa é a dificuldade considerada a seguir.

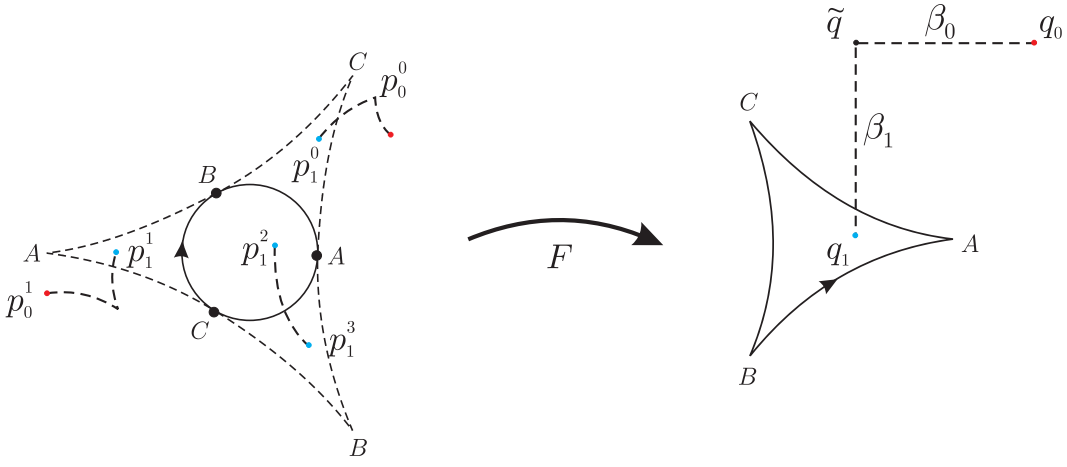



Figura 5.8: Obtendo todas as pré-imagens de pontos remotos

### 5.4.2
### Escolhendo caminhos e invertendo

É inevitável, entretanto, que o caminho em "L" ligando um ponto resolvido $q_0$ a um ponto a inverter $q_1$ cruze às vezes o conjunto $F(\mathcal{C})$. Mais precisamente, suponha que um segmento orientado $\beta = \overline{\alpha\omega}$ desse caminho L encontre $F(\mathcal{C})$ num ponto $q_*$, que divide $\beta$ em dois segmentos menores, $\overline{\alpha q_*}$ e $\overline{q_*\omega}$. O número de pré-imagens de $\overline{\alpha q_*} - q_*$ e o de $\overline{q_*\omega} - q_*$ diferem de 2, pelo Teorema 3.6. Quando o número de pré-imagens diminui, é porque um

par de arcos no domínio obtido por inversão de $\overline{\alpha q_*} - q_*$ estão convergindo para o mesmo ponto $p_*$ no domínio, para o qual vale $F(p_*) = q_*$. O programa identifica esses arcos e interrompe a inversão de $\beta$ para esses arcos.

Quando o número de pré-imagens aumenta, é porque dois novos arcos devem surgir na inversão de $\overline{q_*\omega} - q_*$. Para identificá-los, o programa encontra pontos críticos cujas imagens estão próximas a $q_*$ e, por um refinamento tipo Newton, obtém $p_*$ para o qual $F(p_*) = q_*$. Como cúspides foram evitadas, $p_*$ é uma dobra. Agora, a partir de $p_*$, a análise numérica faz uso da forma local das dobras, implementada na rotina *beget_preim()*. O método é mais sutil do que simplesmente uma iteração do método de Euler-Newton: a condição inicial para a continuação da inversão de $\overline{q_*\omega} - q_*$ não é um ponto onde a Jacobiana da função é inversível.

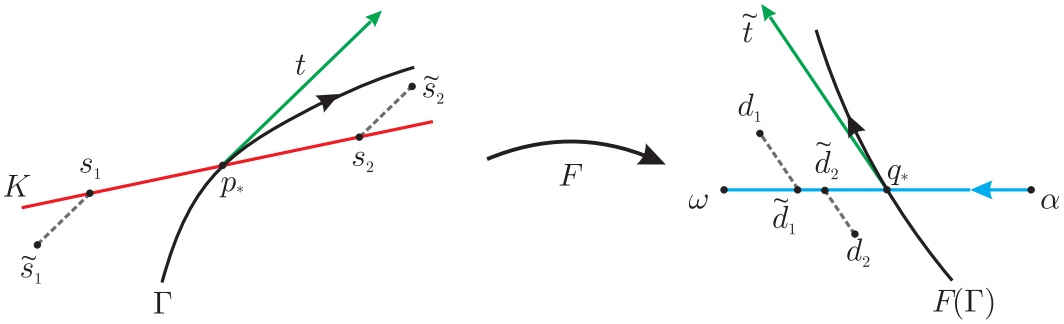



Figura 5.9: Nascem duas pré-imagens

O programa calcula o núcleo $K$ de $DF(p_*)$, indicado em vermelho na figura acima, e o vetor tangente $t$ à curva crítica $\Gamma$, e escolhe dois pontos $s_1$ e $s_2$ sobre a reta $p_* + K$ próximos a $\mathcal{C}$, um de cada lado de $\Gamma$. Pela expansão de Taylor de $F$ em torno de $p_*$,

$$F(s_i) = F(p_*) + \underbrace{DF(p_*)[s_i - p_*]^T}_{0,\text{ pois } p_* \in K} + O(||s_i - p_*||^2) \approx q_*, \quad i = 1, 2.$$

Assim, se $s_i$ distam $\epsilon$ de $p_*$, as imagens $d_i = F(s_i)$ distam $O(\epsilon^2)$ de $q_*$ e estão do mesmo lado de $F(\Gamma)$, mas não estão em $\beta$. Queremos encontrar pontos $\tilde{s}_i$ próximos a $s_i$ cujas imagens estejam de fato em $\beta$. Usar $s_i$ como condição inicial para um método iterativo revelou-se instável numericamente. Em vez disso, o programa escolhe uma condição inicial mais refinada, descrita a seguir. Note que o vetor $\tilde{t} = DF(p_*)t$ é tangente a $F(\Gamma)$. Determine $\lambda_i$ tal que o ponto $\tilde{d}_i = d_i + \lambda_i \tilde{t}$ pertença a $\beta$. Agora, use $\tilde{s}_i = s_i + \lambda_i t$ como condição inicial para resolver $F(\tilde{s}_i) = \tilde{d}_i$.

Finalmente, o processo de inversão de $\overline{q_*\omega} - q_*$ é substituído pela inversão de $\overline{\tilde{d}_i\omega}$, com condições iniciais $\tilde{s}_i$.

Agora, que vimos como o programa procede ao inverter segmentos que trespassam $F(\mathcal{C})$ em dobras, voltamos à escolha de caminhos em forma de L apropriados. Temos que ligar um ponto $q_1$ na imagem de $F$, cujas pré-imagens queremos calcular, a um ponto resolvido $q_0$. O caminho, orientado de $q_0$ a $q_1$, é formado por dois segmentos $\beta_0$ e $\beta_1$, sendo que um é horizontal e outro, vertical. A Figura 5.8 exibe um tal caminho e suas pré-imagens para a função $\widetilde{F}(z) = z^2 + \bar{z}$. O ponto $q_0$ é obtido no banco de pontos resolvidos, entre os *Number of neigh* (um parâmetro global) mais próximos de $q_1$. Para escolher entre esses candidatos, consideram-se as seguintes propriedades:

(a) os segmentos devem passar longe de cúspides e de extremos locais das coordenadas $x$ e $y$ em $F(\mathcal{C})$;

(b) ambos os segmentos devem ter poucas intersecções com $F(\mathcal{C})$;

(c) segmentos curtos são preferidos.

A inversão de pontos próximos a cúspides e a extremos locais pode gerar instabilidade numérica no método. Assim, as condições (a) e (c) são naturais. Aqui, a distância entre pontos é ainda definida pela norma sup, isto é, pela soma dos segmentos do L. A rotina *find_L()* é empregada para selecionar o caminho adequado considerando as propriedades acima. Essa rotina faz chamadas às subrotinas *x_censor()* e *y_censor()* para verificar se os segmentos de reta $\beta_0$ e $\beta_1$ estão próximos de cúspides ou de extremos locais, e também às subrotinas *x_to_ints()* e *y_to_ints()* para calcular, ordenar e contar as intersecções entre segmentos $\beta$ e o conjunto $F(\mathcal{C})$.

Se a inversão do segmento $\beta$ ocorreu com sucesso, decidimos se $q_1$ é incorporado ao banco de pontos resolvidos, verificando o valor do parâmetro global *Banksys*. Se este valor for $-1$, $q_1$ e suas pré-imagens são incorporados; se for 1, verificamos se houve intersecção de $\beta$ com $F(\mathcal{C})$ e somente em caso afirmativo armazenamos $q_1$; se *Banksys* for 0, nada é armazenado. Percebemos aqui a possibilidade de mantermos pontos resolvidos em várias placas de $F(\mathcal{C})$ que poderão se úteis na inversão de pontos situados nessas placas. O banco de pontos é implementado com a estrutura de dados *fila* (*FIFO - First In, First Out*) possuindo um tamanho fixo e, quando o banco está cheio, ao adicionarmos um novo ponto removemos o ponto mais antigo.

Resumindo, para proceder à inversão do ponto $q_1$, escolhemos um L apropriado em *find_L()* e realizamos a inversão por continuação ao longo do L, levando em conta as alternativas a considerar no caso de um segmento encontrar $F(\mathcal{C})$.

### 5.4.3
### Calculando a flor

A flor é o conjunto $\mathcal{F} = F^{-1}(F(C))$. Para gerá-la, o **wx2x2** considera os *arcos maximais* de $F(C)$, ou seja, as componentes conexas de $F(C)$ menos os pontos de intersecção e imagens de cúspides. A rotina *make_flor()* percorre todas as curvas que compõem $F(C)$ e envia cada arco maximal da curva à rotina *make_pre_arc()* — responsável por inverter arcos maximais.

A rotina *make_pre_arc()* tem parâmetros inteiros que informam os índices da imagem de curva crítica e dos pontos inicial e final do arco maximal, respectivamente, $j_{crv}$, $j_a$ e $j_z$. A partir daí, seleciona o ponto $P_m$ de índice médio ($j_m = (j_a + j_z)/2$) e computa um ponto $P$ perto de $P_m$, mas do lado da curva que possui menos pré-imagens. Calcula-se as $k$ pré-imagens de $P$, e pelo Teorema 3.6, sabemos que $P_m$ possui $k + 1$ pré-imagens. Uma das pré-imagens de $P_m$ já é conhecida: é o ponto de índice $j_m$ na curva crítica $j_{crv}$. Para encontrar as outras $k$ pré-imagens de $P_m$, realiza-se uma inversão ao longo do segmento de reta que liga $P$ à $P_m$ para cada pré-imagem de $P$ como condição inicial. Agora, inicia-se a inversão do arco maximal em $P_m$ e inverte-se cada uma de suas metades. A Figura 5.10 ilustra o processo de criação de $\mathcal{F}$ para a função $\widetilde{F}(z) = z^2 + \bar{z}$.

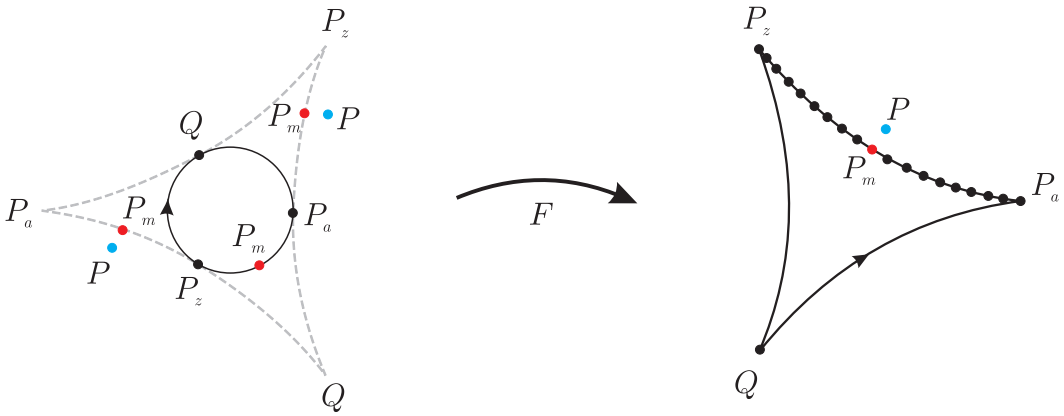



Figura 5.10: Invertendo um arco maximal

A análise numérica torna-se especialmente delicada quando os pontos a inverter se aproximam de cúspides. Flores são mais fáceis de calcular a partir de grandes bancos de pontos resolvidos. Freqüentemente, a flor de uma função é visualmente muito complicada: nem sempre vale a pena calculá-la. Por outro lado, é confortador que o processo de inversão de pontos não depende da construção da flor.