

CPS 250 - Assignment 4 - Spring 2004

David Williams, Jadrian Miles, Jonathan Perlstein

I. NEWTON DIFFERENTIAL EQUATION FOR THE STEEP-VALLEY FUNCTION

A. Background Theory

We wish to solve for \mathbf{x} such that $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ or a local minimum. To do this, we use the Newton differential equation

$$\mathbf{f}'(\mathbf{x}(t))\dot{\mathbf{x}}(t) = -\mathbf{f}(\mathbf{x}(t)) \quad (1)$$

$$\dot{\mathbf{x}}(t) + (\mathbf{f}'\mathbf{x}(t))^{-1}\mathbf{f}(\mathbf{x}(t)) = \mathbf{0}, \mathbf{x}(0) = \mathbf{x}_0. \quad (2)$$

For the steep valley system, with $k = 1$,

$$\mathbf{f} = \begin{bmatrix} (x_1 - x_2^7)^3 + x_2^3 \\ (x_1 + x_2)^2 + 1 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \quad (3)$$

and

$$\mathbf{f}' = \begin{bmatrix} 3(x_1 - x_2^7)^2 & -21x_2^6(x_1 - x_2^7) + 3x_2^2 \\ \frac{0.2}{|x_1 + x_2|^{0.8}} & \frac{0.2}{|x_1 + x_2|^{0.8}} \end{bmatrix} = \begin{bmatrix} b & c \\ a & a \end{bmatrix} = \mathbf{J}. \quad (4)$$

We assert that

$$\mathbf{f}'^{-1} = \frac{1}{a(b-c)} \begin{bmatrix} a & -c \\ -a & b \end{bmatrix} = \tilde{\mathbf{I}}. \quad (5)$$

If this is true, $\mathbf{f}'\tilde{\mathbf{I}} = \mathbf{I}$ and $\tilde{\mathbf{I}}\mathbf{f}' = \mathbf{I}$. We show that

$$\mathbf{f}'^{-1}\tilde{\mathbf{I}} = \begin{bmatrix} b & c \\ a & a \end{bmatrix} \frac{1}{a(b-c)} \begin{bmatrix} a & -c \\ -a & b \end{bmatrix} = \frac{1}{a(b-c)} \begin{bmatrix} ab - ac & bc - bc \\ aa - aa & ab - ac \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (6)$$

and

$$\tilde{\mathbf{I}}\mathbf{f}' = \frac{1}{a(b-c)} \begin{bmatrix} a & -c \\ -a & b \end{bmatrix} \begin{bmatrix} b & c \\ a & a \end{bmatrix} = \frac{1}{a(b-c)} \begin{bmatrix} ab - ac & ac - ac \\ ba - ab & ab - ac \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad (7)$$

which proves that $\tilde{\mathbf{I}} = \mathbf{f}'^{-1}$ is indeed true. Now we want to solve $\mathbf{f}'\mathbf{y} + \mathbf{f} = \mathbf{0} \rightarrow \mathbf{y} = \mathbf{f}'^{-1}(-\mathbf{f})$ for \mathbf{y} :

$$\mathbf{y} = \mathbf{f}'^{-1}(-\mathbf{f}) = \frac{-1}{a(b-c)} \begin{bmatrix} a & -c \\ -a & b \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \quad (8)$$

$$= \frac{-1}{a(b-c)} \begin{bmatrix} af_1 - cf_2 \\ bf_2 - af_1 \end{bmatrix} = \frac{1}{a(b-c)} \begin{bmatrix} -af_1 + cf_2 \\ af_1 - bf_2 \end{bmatrix}. \quad (9)$$

Verification of the above is shown by

$$\begin{aligned} \mathbf{f}'\mathbf{y} + \mathbf{f} &= \begin{bmatrix} b & c \\ a & a \end{bmatrix} \frac{1}{a(b-c)} \begin{bmatrix} -af_1 + cf_2 \\ af_1 - bf_2 \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \\ &= \frac{1}{a(b-c)} \begin{bmatrix} -abf_1 + bcf_2 + acf_1 - bcf_2 \\ a^2f_1 + acf_2 + a^2f_1 - abf_2 \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \\ &= \frac{1}{a(b-c)} \begin{bmatrix} a(c-b)f_1 \\ a(c-b)f_2 \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \\ &= \begin{bmatrix} -f_1 \\ -f_2 \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \mathbf{0}. \end{aligned} \quad (10)$$

We can further simplify \mathbf{y} as

$$\mathbf{y} = \frac{1}{a(b-c)} \begin{bmatrix} -af_1 + cf_2 \\ af_1 - bf_2 \end{bmatrix} = \frac{1}{b-c} \begin{bmatrix} -f_1 + \frac{c}{a}f_2 \\ f_1 - \frac{b}{a}f_2 \end{bmatrix}. \quad (11)$$

Along $x_1 + x_2 = 0$, $a = \infty$, and so

$$\mathbf{y} = \frac{1}{b-c} \begin{bmatrix} -f_1 \\ f_1 \end{bmatrix} \text{ for } x_1 + x_2 = 0. \quad (12)$$

From (5), we see that the Jacobian \mathbf{J} is singular anywhere $b = c$. The normal equations for $\mathbf{f}'\mathbf{y} = -\mathbf{f}$ when \mathbf{J} is singular are (as defined in §2.7.2)

$$\begin{aligned}
\mathbf{f}'^T \mathbf{f}' \mathbf{y} &= (\mathbf{f}'^T)(-\mathbf{f}) \\
\begin{bmatrix} b & a \\ c & a \end{bmatrix} \begin{bmatrix} b & c \\ a & a \end{bmatrix} \mathbf{y} &= \begin{bmatrix} b & a \\ c & a \end{bmatrix} \begin{bmatrix} -f_1 \\ -f_2 \end{bmatrix} \\
\begin{bmatrix} b^2 + a^2 & bc + a^2 \\ bc + a^2 & c^2 + a^2 \end{bmatrix} \mathbf{y} &= \begin{bmatrix} -bf_1 - af_2 \\ -cf_1 - af_2 \end{bmatrix} \\
(a^2 + b^2) \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \mathbf{y} &= - \begin{bmatrix} bf_1 + af_2 \\ bf_1 + af_2 \end{bmatrix} \\
\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} &= \frac{-bf_1 - af_2}{a^2 + b^2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
\begin{bmatrix} y_1 + y_2 \\ y_1 + y_2 \end{bmatrix} &= \frac{-bf_1 - af_2}{a^2 + b^2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\end{aligned} \tag{13}$$

so $y_1 + y_2 = -\left(\frac{bf_1 + af_2}{a^2 + b^2}\right)$. So

$$\mathbf{y} = \mathbf{z}(p) = \begin{bmatrix} pr \\ (1-p)r \end{bmatrix} \text{ for } r = -\left(\frac{bf_1 + af_2}{a^2 + b^2}\right) \tag{14}$$

satisfies $y_1 + y_2 = r$ for any p . Then we can write

$$|\mathbf{z}(p)|^2 = p^2 r^2 + (1-p)^2 r^2, \tag{15}$$

so $\frac{d}{dp} |\mathbf{z}(p)|^2 = 2pr^2 - 2(1-p)r^2 = 0$ when $p = \frac{1}{2}$. Since $\frac{d^2}{dp^2} |\mathbf{z}(p)|^2 = 2r^2 + 2r^2 > 0$, $p = \frac{1}{2}$ minimizes $|\mathbf{z}(p)|^2$, and therefore minimizes $|\mathbf{z}(p)|$ as well. Substituting $\mathbf{y} = \mathbf{z}(p)$ into $\mathbf{f}'\mathbf{y} + \mathbf{f}$, we obtain

$$\begin{aligned}
\mathbf{f}'\mathbf{z}(p) + \mathbf{f} &= \begin{bmatrix} b & c \\ a & a \end{bmatrix} - \left(\frac{bf_1 + af_2}{a^2 + b^2}\right) \begin{bmatrix} p \\ 1-p \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \\
&= -\left(\frac{bf_1 + af_2}{a^2 + b^2}\right) \begin{bmatrix} pb + (1-p)c \\ pa + (1-p)a \end{bmatrix}
\end{aligned}$$

$$= r \begin{bmatrix} pb + (1-p)c \\ a \end{bmatrix}. \quad (16)$$

So

$$\|\mathbf{f}'\mathbf{z}(p) + \mathbf{f}\|^2 = \|r\|^2(a^2 + (pb + (1-p)c)^2) = \|r\|^2(a^2 + p^2b^2 + 2pbc - 2p^2bc + (1-p)^2c^2) \quad (17)$$

and when $b = c$

$$\frac{d}{dp}\|\mathbf{f}'\mathbf{z}(p) + \mathbf{f}\|^2 = \|r\|^2(2pb^2 + 2bc - 4pbc - 2(1-p)c^2) = \|r\|^2(2pb^2 + 2b^2 - 4pb^2 - 2(1-p)b^2) = 0 \quad (18)$$

for any p , and

$$\frac{d^2}{dp^2}\|\mathbf{f}'\mathbf{z}(p) + \mathbf{f}\|^2 = \|r\|^2(2b^2 - 4b^2 + 2b^2) = 0 \quad (19)$$

for any p as well. Therefore, $\mathbf{y} = \mathbf{z}(\frac{1}{2})$ is the least squares solution.

Next let $h^2(t) = \mathbf{f}(\mathbf{x}(t))^T \mathbf{f}(\mathbf{x}(t))$, which implies $h(t) = \|\mathbf{f}(\mathbf{x}(t))\|$. Differentiating $h^2(t)$, we see that $2h\dot{h} = 2\mathbf{f}^T \mathbf{f}'\dot{\mathbf{x}} = 2\mathbf{f}^T \mathbf{f}'\mathbf{y}$ along $x_1 + x_2 = 0$. Therefore, $h\dot{h} = \mathbf{f}^T \mathbf{f}'\mathbf{y} = \mathbf{f}^T(-\mathbf{f}) = -\mathbf{f}^T \mathbf{f} = -h^2$, so $\dot{h} = -h$. This is a differential equation, and solving gives

$$h(\mathbf{x}(t)) = e^{-t}h_0. \quad (20)$$

However, this applies only when $\mathbf{f}'\mathbf{y} + \mathbf{f} = \mathbf{0}$, which is only true for non-singular \mathbf{f}' . If we consider *singular* \mathbf{f}' , then we use $\mathbf{y} = \mathbf{z}(\frac{1}{2}) = \frac{1}{2} \begin{bmatrix} r \\ r \end{bmatrix}$, and so

$$h\dot{h} = \mathbf{f}^T \mathbf{f}'\mathbf{y} = \begin{bmatrix} f_1 & f_2 \end{bmatrix} \begin{bmatrix} b & c \\ a & a \end{bmatrix} - \left(\frac{bf_1 + af_2}{2(a^2 + b^2)}\right) \begin{bmatrix} 1 \\ 1 \end{bmatrix} = -\left(\frac{bf_1 + af_2}{2(a^2 + b^2)}\right) (bf_1 + af_2 + cf_1 + af_2). \quad (21)$$

Letting $b = c$, we simplify by writing

$$h\dot{h} = \left(\frac{-2(bf_1 + af_2)^2}{2(a^2 + b^2)}\right) = \left(\frac{-(bf_1 + af_2)^2}{(a^2 + b^2)}\right) < 0. \quad (22)$$

But

$$\mathbf{f}'^T \mathbf{f} = \begin{bmatrix} b & a \\ c & a \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} bf_1 + af_2 \\ cf_1 + af_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (23)$$

implies $\mathbf{y} = \mathbf{0}$, a local minimum or $\mathbf{f} = \mathbf{0}$. In this case, $h\dot{h} = 0$, which implies $-h \leq \dot{h} < 0$ and thus h decays more slowly, when \mathbf{f}' is singular, than indicated by (20).

B. Newton Direction Calculations

We have now derived three different forms of the Newton direction y : one for use under normal conditions, another for when the Jacobian is singular, and a third for when elements of the Jacobian itself are undefined. These forms appear, respectively, in equations (11), (14), and (12), and are summarized below:

$$\text{Normal conditions: } \mathbf{y} = \frac{1}{a(b-c)} \begin{bmatrix} -af_1 + cf_2 \\ af_1 - bf_2 \end{bmatrix} \quad (24)$$

$$\text{Singular Jacobian (for } b-c=0\text{): } yv = \begin{bmatrix} r/2 \\ r/2 \end{bmatrix} \text{ for } r = -\left(\frac{bf_1 + af_2}{a^2 + b^2}\right) \quad (25)$$

$$\text{Undefined Jacobian (for } x_1 + x_2 = 0\text{): } \mathbf{y} = \frac{1}{b-c} \begin{bmatrix} -f_1 \\ f_1 \end{bmatrix} \quad (26)$$

In a computer implementation one must consider that the problems with the Jacobian arising when these conditions are met exactly also manifest themselves in regions around the condition curves. Therefore, we apply the singular Jacobian equation when $|b-c| < 10^{-6}$ and the undefined Jacobian equation when $|x_1 + x_2| < 10^{-6}$. For ease of future reference, these regions will be designated SJ and UJ, respectively.

A fourth region arises when we place intervals around the condition curves, however: the intersection of SJ and UJ has its own special properties. When the regions overlap, neither equation for the Newton direction is valid, as each one relies on the other's conditions being false.

Furthermore, consider a situation in which $x_1 + x_2 = 0$. Adding the Newton direction to the position preserves the condition; our equation traps points in UJ as soon as they enter:

$$x^{(i+1)} = x^{(i)} + y = \begin{bmatrix} x_1^{(i)} - \frac{f_1}{b-c} \\ x_2^{(i)} + \frac{f_1}{b-c} \end{bmatrix}. \quad (27)$$

So if $x_1^{(i)} + x_2^{(i)} = 0$, then $x_1^{(i+1)} + x_2^{(i+1)} = x_1^{(i)} - \frac{f_1}{b-c} + x_2^{(i)} + \frac{f_1}{b-c} = 0$ as well.

Note, however, that this Newton direction draws points within UJ toward the origin, where $b-c=0$; any iteration path that falls into the UJ will eventually be drawn into the overlap region and will not escape it.

To deal with the overlap, we note that $x_1 + x_2 = 0$ is a straight line in the x_1, x_2 plane with slope -1 . To allow a trapped path to escape the overlap region, then, we wish to travel orthogonal to this line, but there are two choices of direction: up and right, or down and left. To make the decision, we examine the behavior of the Newton direction defined for SJ along the line $x_1 = x_2$ near the origin. $f = [0; 1]$ at the origin, and so $r = -(bf_1 + af_2)/(a^2 + b^2)$ is a large negative value on both sides of the origin, since $a \rightarrow \infty$. Therefore the Newton direction to be used in the SJ,UJ intersection region should be some $y = [-\alpha; -\alpha]$; we choose $\alpha = 1$ and allow damping to choose the proper scaling factor.

C. Results

We utilized Newton's method on the given system of nonlinear equations, without any refinements such as two-parameter damping or punting. Instead, we allowed the function to take its course and converge to wherever the bare algorithm caused determined.

Figure 1 shows the x and y coordinates to which various initial starting points converge. The range of initial starting points is from -10 to 10 in both the x and y directions, with a grid of over $10,000$ initial starting points in all.

In order to save calculations while creating these plots, we attempted to recycle information from previous calculations. In the course of determining the convergence path from a particular starting position, if a point is found to be surrounded on all sides by points that all converge to the same position, it is reasonable to assume that it will also converge to this same position in approximately the same number of additional iterations. Therefore we chose starting positions in a spiral pattern outward from the origin, checking after each iteration of each point's convergence calculation whether it had entered a region of consistent convergence. If the point's four closest neighbors in space all converge to the same (within a small tolerance) position, then the convergence calculation is short-circuited, the same position is reported as the current start point's convergence position, and the median number of iterations it took for those neighboring points to converge is added to the number of iterations it took for the point to arrive in the region, and this is taken as the number of iterations to convergence. This method greatly increases the speed of our test, as thousands of points (especially in the lower half of the plane) may be short-circuited after very few iterations.

Figure 2 shows the number of iterations needed to converge to either the root or a local minimum, for various initial starting points.

This figure is dominated by two different bands, on which the initial points tend to converge very quickly. The first band is along $x_1 + x_2 = 0$. The second band is from approximately $y = -2$ to $y = 2$. Note that different points within the same band may converge to different minima, but they will converge within approximately the same number of iterations. The y value of the initial point has a much greater effect on the number of iterations until convergence than the x value does. As the magnitude of y increases, the number of iterations needed to converge also increases. This behavior is seemingly independent of x. The noticeable band running through the origin with a slope of -1 coincides with the different calculation of the Newton direction used in the region near $x_1 + x_2 = 0$; every point within this region attracts to the origin at a consistent rate and then converges in an equal number of iterations to its destination.

Figure 3 presents the results of Figure 1 in a manner such that the point of convergence for each starting point can be compared more easily. In this figure, points colored in red correspond to points that converged to the root, at $x = 0, y = -1$. Points colored in yellow correspond to points that converged to the local minimum near $x = -0.5, y = -0.5$. Points colored in light blue also correspond to points that converged to a local minimum near $x = -0.4, y = -0.4$.

Approximately half of the initial starting points converge to the actual root. Generally, initial points with a negative y values lead to convergence to the root. A scattering of initial points with positive y values also converge to the root. Interestingly, the points in the top half of the x-y plane that do converge to the root are distributed in a nearly random fashion. The majority of the rest of the points with initial positive y values converge to the same local minimum. A small number of other points converge to a different local minimum. While these points generally have an initial x value close to zero, their distribution also appears to be highly irregular.

The unpredictable convergence to local minima indicates that even with specialized Newton direction calculations for ill-conditioned regions of our system, it is still necessary to implement a punting algorithm in order to reliably escape local minima and converge to the root.

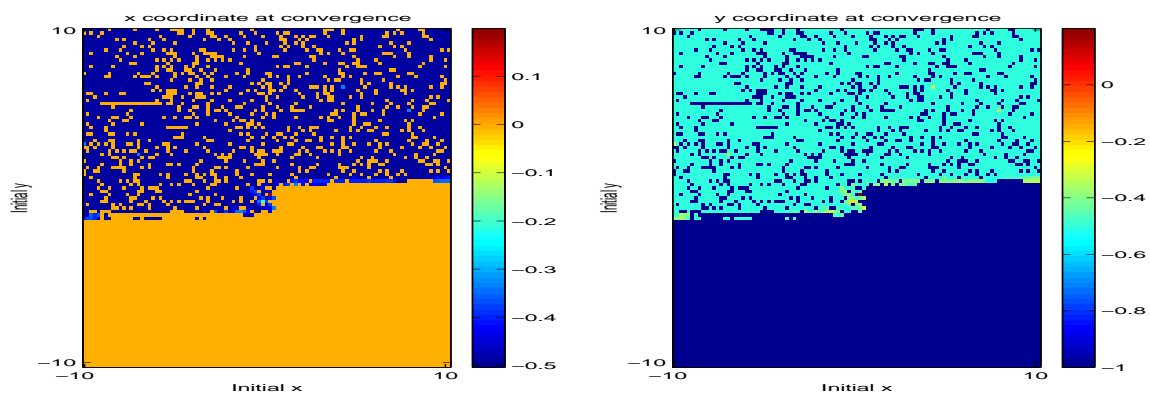


Fig. 1. Coordinates to which initial starting points converge.

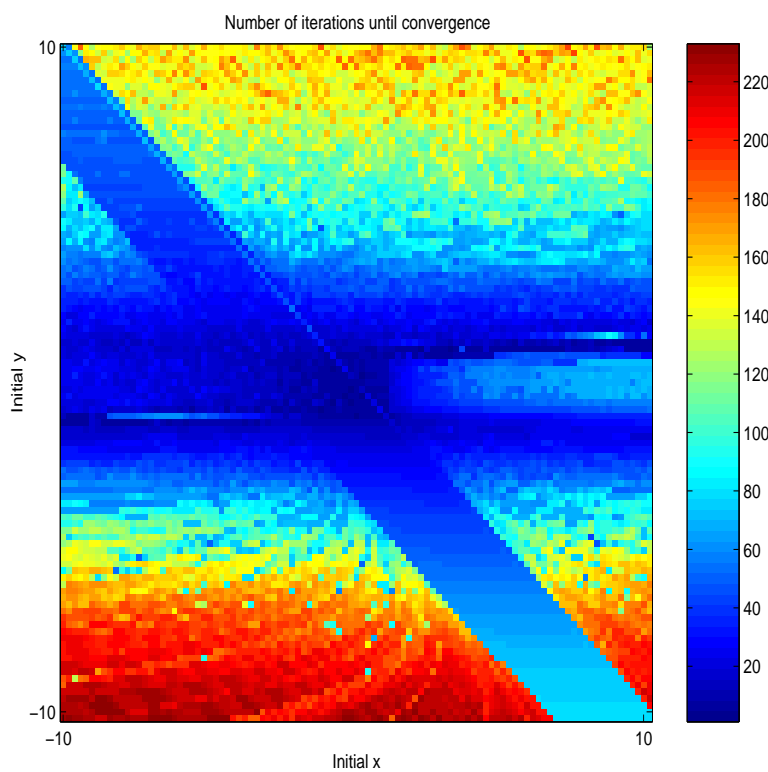


Fig. 2. Number of iterations until initial starting points converge.

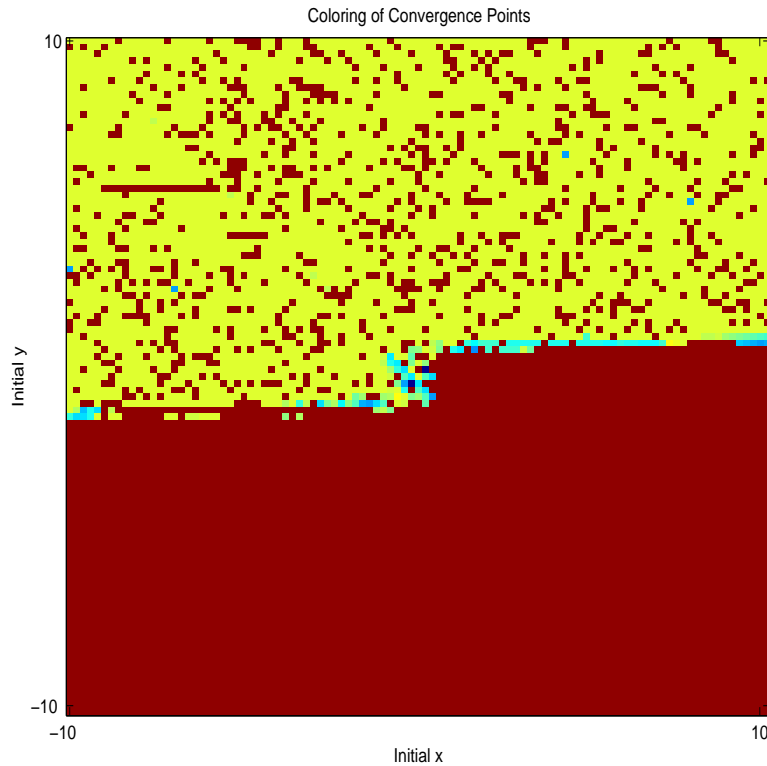


Fig. 3. Unique coloring of convergence points for initial starting points.

II. CONSTANT-J SECOND-ORDER ODE BVP SOLVER

A. Background Theory

We are presented with the boundary value problem

$$-j'(x) + u(x) = b(x) = 100e^{(-64(x-3/4)^2)} \quad (28)$$

$$j(x) = u'(x) + \beta u(x) \quad (29)$$

$$u(0) = u(1) = 0 \quad (30)$$

where derivatives are with respect to x .

We wish to solve this ODE for u over the region $x \in [0, 1]$ with various values of β (positive and negative, on the order 0 to 500). Through traditional techniques of ordinary differential equations, as described in section 4.6, we arrive at an iterative method for refining discretized

approximations of u . In each iteration of this method, the current u approximation leads to a tridiagonal matrix \mathbf{T} , for which solving

$$\mathbf{T}u = b(x) \quad (31)$$

defines a refined u for all points x . This next iteration's x includes all the previous discretization positions, as well as all the midpoints of the intervals between them.

We adopt the following indexing notation. Quantities represented with a subscript $i+$ denote the region to the right of the point i , whereas a subscript $i-$ denotes the region to the left of the point i .

For the constant current assumption, we assume in (29) j_{i+} and j_{i-} are constant on the intervals (x_i, x_{i+1}) and (x_{i-1}, x_i) , respectively. From (29) we multiply by $e^{\beta x}$ to obtain

$$(e^{\beta x} u)' = j e^{\beta x}. \quad (32)$$

From this equation, significant manipulation gives the following series of equations

$$(e^{\beta x} u)' = (\beta u_i + u') e^{\beta x} \quad (33)$$

$$(e^{\beta x} u)' = \left(\beta u_i + \frac{u_{i+1} - u_i}{x_{i+1} - x_i} \right) e^{\beta x} \quad (34)$$

$$(x_{i+1} - x_i)(e^{\beta x} u)' = (\beta u_i(x_{i+1} - x_i) + u_{i+1} - u_i) e^{\beta x} \quad (35)$$

$$(x_{i+1} - x_i)(e^{\beta x} u)' = \left(u_i(x_{i+1} - x_i) + \frac{u_{i+1} - u_i}{\beta} \right) (e^{\beta x})' \quad (36)$$

$$(x_{i+1} - x_i)(e^{\beta x} u)' = \left(u_i + \frac{u_{i+1} - u_i}{\beta(x_{i+1} - x_i)} \right) (x_{i+1} - x_i)(e^{\beta x})' \quad (37)$$

$$e^{\beta x_{i+1}} u_{i+1} - e^{\beta x_i} u_i = \left(\frac{j_{i+}}{\beta} \right) (e^{\beta x_{i+1}} - e^{\beta x_i}). \quad (38)$$

Next by multiplying by $e^{-\beta x_i}$, and defining $h_{i+} = x_{i+1} - x_i$ we obtain

$$e^{\beta h_{i+}} u_{i+1} - u_i = \left(\frac{j_{i+}}{\beta} \right) (e^{\beta h_{i+}} - 1). \quad (39)$$

Letting $l_{i+} = \beta h_{i+}$, we obtain

$$l_{i+} \left(\frac{e^{l_{i+}} u_{i+1} - u_i}{e^{l_{i+}} - 1} \right) = h_{i+} j_{i+}. \quad (40)$$

Defining the Bernoulli function

$$B(x) = \frac{x}{e^x - 1}, \quad (41)$$

we note that

$$B(x) + x = e^x B(x) = B(-x) \quad (42)$$

so that

$$j_{i+} = \left(\frac{l_{i+}}{h_{i+}} \right) \left(\frac{e^{l_{i+}u_{i+1}} - u_i}{e^{l_{i+}} - 1} \right) \quad (43)$$

$$-j_{i+} = \left(\frac{l_{i+}}{h_{i+}} \right) \left(\frac{u_i - e^{l_{i+}u_{i+1}}}{e^{l_{i+}} - 1} \right) \quad (44)$$

$$-j_{i+} = \left(\frac{1}{h_{i+}} \right) \left(\frac{l_{i+}u_i - l_{i+}e^{l_{i+}u_{i+1}}}{e^{l_{i+}} - 1} \right) \quad (45)$$

$$-j_{i+} = \left(\frac{1}{h_{i+}} \right) (B(l_{i+})u_i - B(-l_{i+})u_{i+1}). \quad (46)$$

Next defining $C(x) = (B(x) + B(-x))/2$, we can note that

$$B(|x|) = C(x) - |x|/2, \quad (47)$$

which in turn allows us to write (46) as

$$-j_{i+} = \left(\frac{1}{h_{i+}} \right) ((C(l_{i+}) - l_{i+}/2)u_i - (C(l_{i+}) + l_{i+}/2)u_{i+1}). \quad (48)$$

Similarly,

$$j_{i-} = \left(\frac{1}{h_{i-}} \right) ((C(l_{i-}) - l_{i-}/2)u_i - (C(l_{i-}) + l_{i-}/2)u_{i-1}). \quad (49)$$

To compute the \mathbf{T} matrix, we of course require the partial derivatives of (48) and (49) with respect to u_{i-1}, u_i , and u_{i+1} . Performing the differentiation, we obtain

$$\frac{\partial(-j_{i+})}{\partial u_i} = \left(\frac{1}{h_{i+}} \right) (C(l_{i+}) - l_{i+}/2) \quad (50)$$

$$\frac{\partial(-j_{i+})}{\partial u_{i+1}} = \left(\frac{-1}{h_{i+}} \right) (C(l_{i+}) + l_{i+}/2) \quad (51)$$

$$\frac{\partial(j_{i-})}{\partial u_{i-1}} = \left(\frac{-1}{h_{i-}} \right) (C(l_{i-}) - l_{i-}/2) \quad (52)$$

$$\frac{\partial(j_{i-})}{\partial u_i} = \left(\frac{1}{h_{i-}} \right) (C(l_{i-}) + l_{i-}/2) \quad (53)$$

We note that the tridiagonal matrix \mathbf{T} will be constructed from (50)-(53). The main diagonal will be the sum of (50) and (53). The diagonal below the main diagonal will be (52). The

diagonal above the main diagonal will be (51). Finally, the main diagonal of \mathbf{T} must also have a contribution from the mean of the intervals to the left and right (i.e. h_{i-} and h_{i+} respectively), because of the presence of u in (28).

It is this tridiagonal system of linear equations, $\mathbf{T}\mathbf{u} = \mathbf{b}$, that we solve for \mathbf{u} , where the components of \mathbf{u} are $u_i = u(x_i)$ for $x_i \in (0, 1)$. We hold the tridiagonal matrix \mathbf{T} as a set of three vectors, each representing one of the diagonals. Since this matrix is tridiagonal and symmetric positive definite, performing gaussian elimination would not require any pivots, so we use a highly optimized tridiagonal solver. Storing the matrices as vectors saves massive amounts of space and using the optimized algorithm saves clock cycles as well.

B. Interval Refinement

Each iteration of our method, we divide each existing interval into two equal subintervals, and then recalculate u at the endpoints of every interval. Proceeding in this way every iteration would result in an evenly-spaced grid of calculation points over the domain of the BVP.

In order to conserve calculations, though, we wish to compute u values only at positions where they provide useful information. That is, we want a variable-sized grid and a method to concentrate interval refinements, and hence calculation positions, in the areas that “need” them in some sense. We quantify this need by estimating a truncation error, the amount by which the j value on an interval in the k -th iteration differs from the values of the two subintervals in the $k + 1$ -th iteration.

Assuming that this truncation error is $O(h)$, we use Taylor’s theorem as part of Richardson extrapolation to derive the following equation:

$$truncerr = |j - j(x)| = \left| j - \frac{j_- + j_+}{2} \right| \quad (54)$$

Where j is the j -value for the original interval, $j(x)$ is the actual j -value at the center of this interval, and j_- and j_+ are the j -values for the left and right subintervals, respectively.

We pick a constant truncation error tolerance to apply across the entire domain. If an interval’s calculated truncation error is higher than the tolerance, its subdivision is preserved for the next iteration; if the truncation error falls below the tolerance, the refinement is ignored. In this way, only subdivisions that provide meaningful data about the j -values are preserved, and u -calculations can be concentrated in regions where they are needed. If the truncation error

tolerance varied across the domain, it would defeat the purpose of differentiating necessary refinements from redundant ones.

The final issue to consider, then, is how to pick the truncation error tolerance. If one is familiar with the scale of the j -values in a particular problem, a tolerance can be chosen from practical experience. The magnitude of the j -values in our problem, however, changes significantly with β as well as over the domain for each β which can be seen in figure 14. Therefore we develop a relative error method: in each iteration of our BVP solver, we calculate the maximum magnitude of the j -values and multiply it by a scaling factor that is passed in as an argument to the function—we used $2 * 10^{-4}$. This value is then capped at a certain cutoff ($4 * 10^{-3}$ in our code), and the resulting value is used as the truncation error tolerance for that iteration. This assures us that the error tolerance will scale with our problem even if the ODE changed significantly, and still guarantees a uniform tolerance across the whole grid within each iteration.

C. Results

Figures 4 through 10 show the solution u versus x , as well as the reciprocal of the interval lengths ($1/h$) versus x , for $\beta = 0, 10, 100, 500, -10, -100, -500$, respectively. It can be seen that the reciprocal of the interval length is large when the interval length is small, which occurs when significant refinement is needed because of relatively more rapidly changing j , both spatially and through iterations, as discussed below.

Figure 11 shows the solution of u versus x for several different tolerances at $\beta = 0$. As the tolerance increases, fewer intervals are needed to converge to a solution, but the solution's form is relatively invariant to the tolerance.

Figure 12 shows the solution of u versus x for several different values of β . We shall describe these plots by comparing them to the $\beta = 0$ case. As β increases above zero, the curve of u shifts to the left, the magnitude decreases, and the peak becomes broader, akin to a plateau. As β decreases below zero, the curves shift to the right, the magnitude decreases, and the peak becomes more prominent and sharper.

Figure 13 shows that the number of intervals increases as $|\beta|$ increases. The BVP is more difficult to solve within tolerance for extreme values of β , so more refinement is necessary to accurately represent the solution.

Figure 14 shows the solution of j versus x for several different values of β . It can be seen that the curve of j shifts up as β increases, but that the shape is retained.

Figures 15 and 16 illustrate the evolution of u as a function of iteration for $\beta = 100$ and $\beta = -100$, respectively. These figures show the refinement of u that occurs before convergence is reached. Note that the function changes most drastically over the course of the iterations around $x = 0.75$; this is why the most refinements are concentrated in this region in figures 4 through 10.

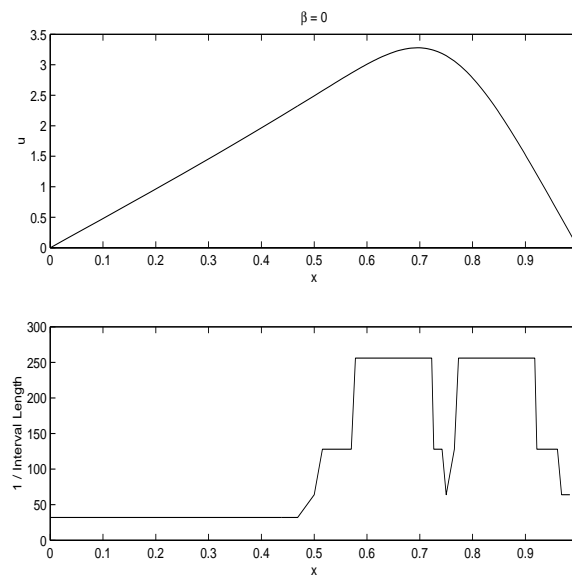


Fig. 4. u and $1 / (\text{interval length})$ at convergence for $\beta = 0$.

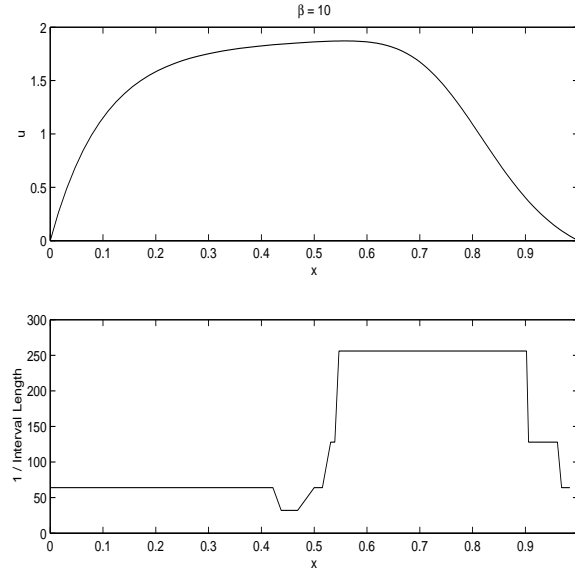


Fig. 5. u and $1 / (\text{interval length})$ at convergence for $\beta = 10$.

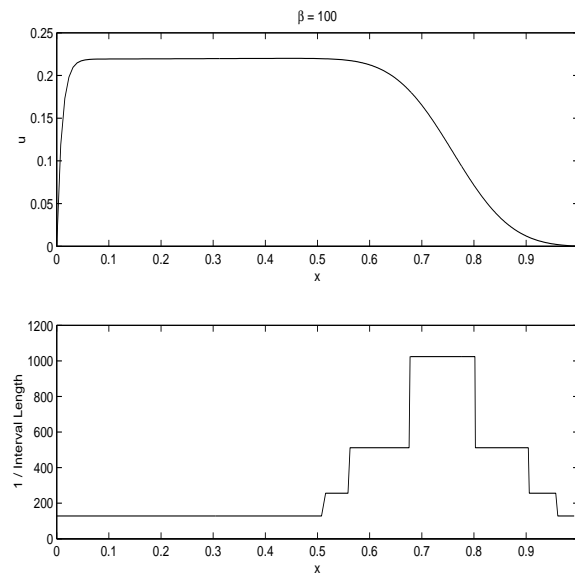


Fig. 6. u and $1 / (\text{interval length})$ at convergence for $\beta = 100$.

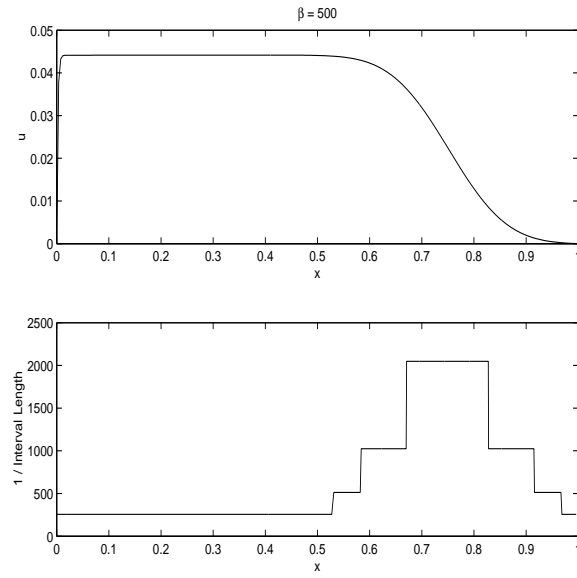


Fig. 7. u and $1 / (\text{interval length})$ at convergence for $\beta = 500$.

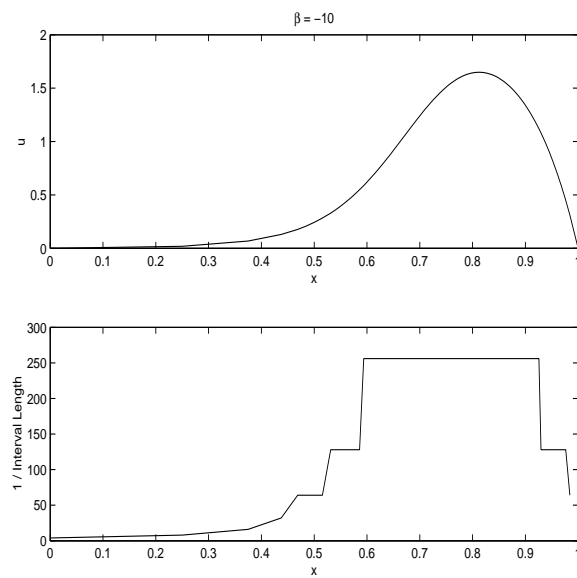


Fig. 8. u and $1 / (\text{interval length})$ at convergence for $\beta = -10$.

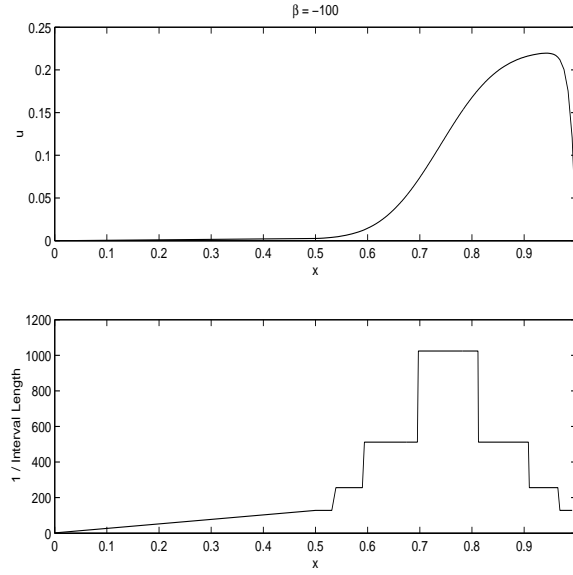


Fig. 9. u and $1 / (\text{interval length})$ at convergence for $\beta = -100$.

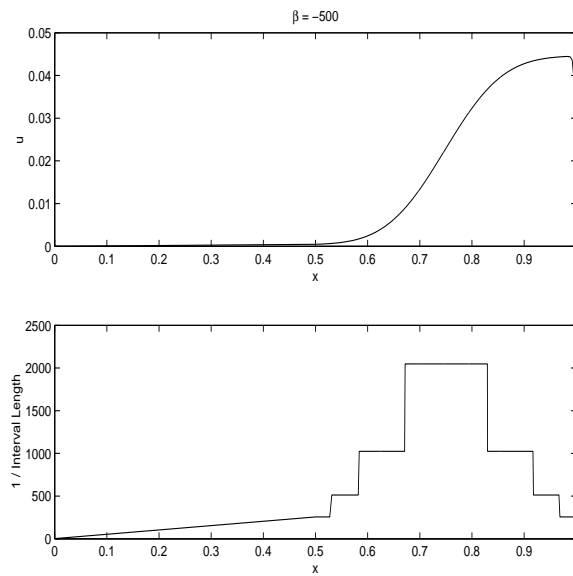


Fig. 10. u and $1 / (\text{interval length})$ at convergence for $\beta = -500$.

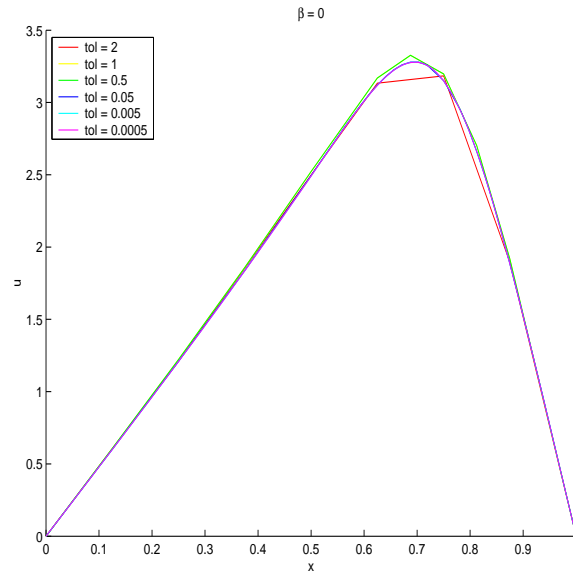


Fig. 11. u at convergence for different tolerances for $\beta = 0$.

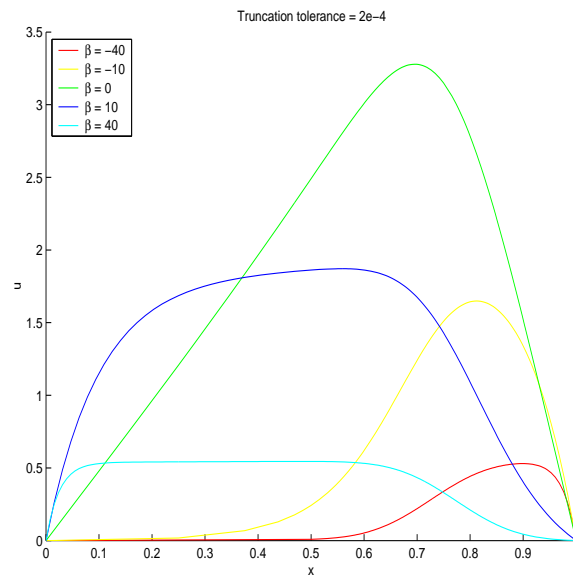


Fig. 12. u at convergence for several values of β .

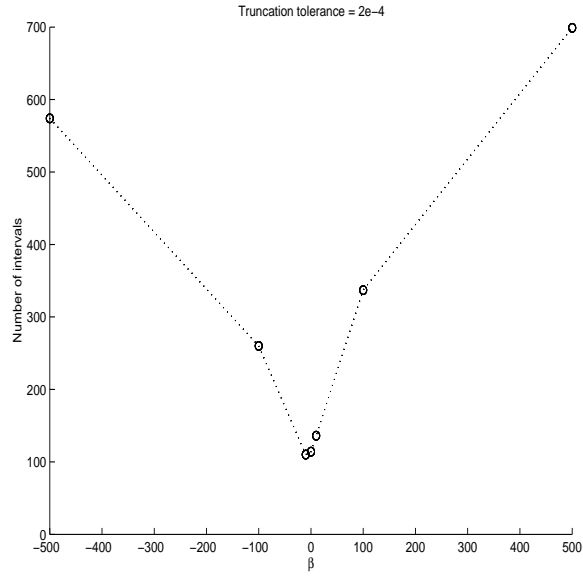


Fig. 13. Number of intervals at convergence for several values of β .

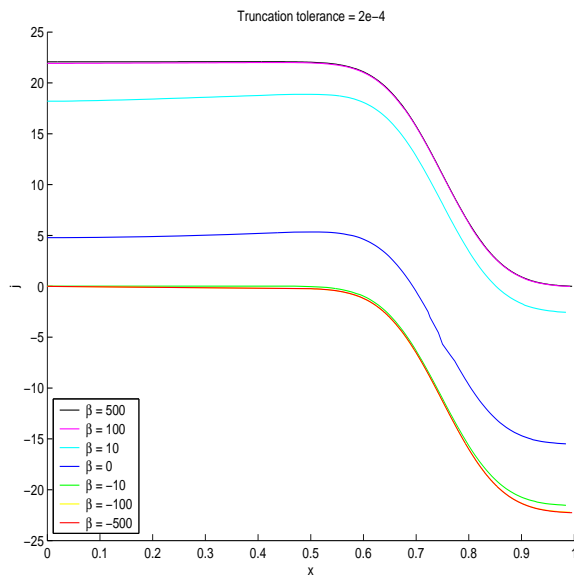


Fig. 14. j at convergence for several values of β .

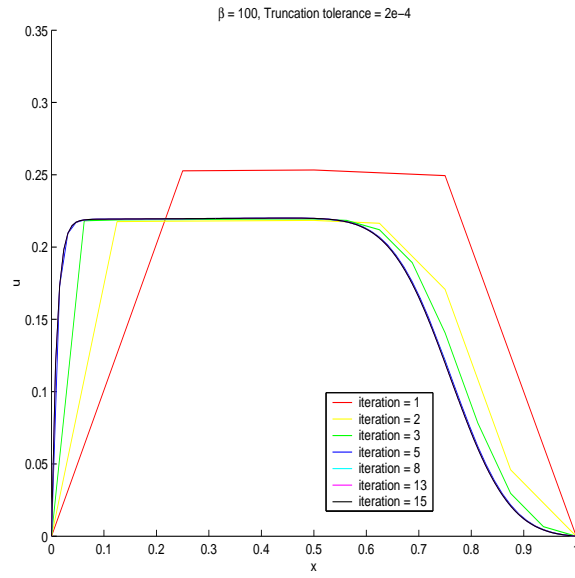


Fig. 15. Evolution of u for $\beta = 100$.

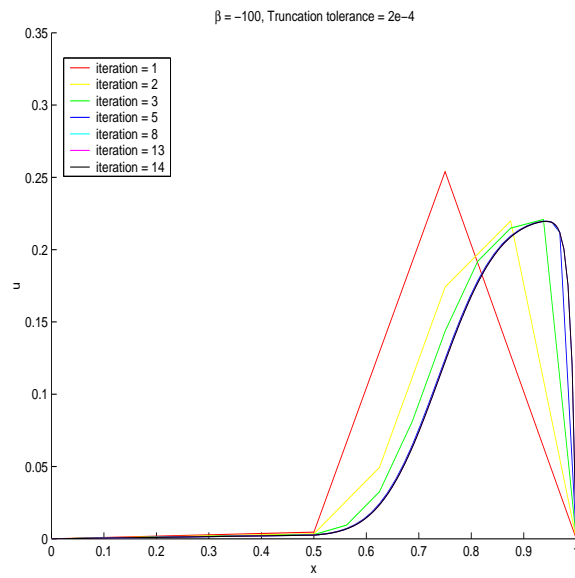


Fig. 16. Evolution of u for $\beta = -100$.

III. APPENDIX

A. Part I. Matlab Code

```

% Nonlinear system Newton's method solver for 2x2 steep valley system with
% attraction tracing and updated nonsingular behavior

function [T] = trace_attr_newton(ULcorner, LRcorner, divs)
    % ULcorner is a column vector indicating the upper-left corner: [x;y]
    % LRcorner is a column vector indicating the lower-right corner: [x;y]
    % divs is a column vector of the number of [horizontal;vertical]
    %     divisions in the grid. T will be divs(2,1) X divs(1,1).

    % T(r,c,:) holds, in order: x position, y position, x and y of attraction point,
    % and an iteration count.
    % Create and initialize T
    T = zeros(divs(2,1),divs(1,1),5);
    for r=1:divs(2,1)
        for c=1:divs(1,1)
            % linearly interpolate position values from UL and LR corners
            T(r,c,1) = ULcorner(1,1) + (c-1)*(LRcorner(1,1) - ULcorner(1,1))/(divs(1,1)-1);
            T(r,c,2) = ULcorner(2,1) - (r-1)*(ULcorner(2,1) - LRcorner(2,1))/(divs(2,1)-1);
            T(r,c,5) = -1;
        end
    end

    % fill the grid by spiraling outward from the center
    runsofar = 0;          % how many steps we have taken on the current side
    direction = [-1;0];   % initial direction is 'up'
    startpos = round(divs ./ 2);
    currpos = startpos;
    maxposnorm = ceil(max(startpos)*sqrt(2));
    while norm(currpos - startpos) < maxposnorm
        % until the spiral covers the whole grid
        if min(currpos) > 0 & currpos <= flipud(divs)
            % calculate on each legal gridpoint
            T = calculate_gridpoint(T, currpos, ULcorner, LRcorner, divs);
        end
        currpos = currpos + direction/norm(direction);
        runsofar = runsofar + 1;
        if runsofar == max(abs(direction))
            % if you've gone far enough, turn right
            if direction(2,1) == 0
                % vertical movement: turning right is a sign change
                direction = -direction;
            else
                % horizontal movement: the next side is longer

```

```

        direction(2,1) = sign(direction(2,1))*(abs(direction(2,1)) + 1);
    end
    % swap components
    temp = direction(1,1);
    direction(1,1) = direction(2,1);
    direction(2,1) = temp;
    runsofar = 0;
end
end

function [T] = calculate_gridpoint(T, currpos, ULcorner, LRCorner, divs)
    stoptol = 1e-6;
    % stoptol is the precision to which we want |g(u,v)| = 0
    % when valleyfinder terminates

    %%%%% BEGIN NEWTON LOOP %%%%%

    u(1,1) = T(currpos(1,1),currpos(2,1),1);
    u(2,1) = T(currpos(1,1),currpos(2,1),2);
    gp = g(u);
    normg = norm(gp);
    k = 1;
    notdone = 1;
    while (normg > stoptol) & notdone
        % solve Jx = -g; x is the Newton direction
        %u
        J = gprime(u); % Jacobian

        %%%%% BEGIN NIFTY SINGULARITY SOLVING %%%%%

        a = J(2,1);
        b = J(1,1);
        c = J(1,2);
        f1 = gp(1,1);
        f2 = gp(2,1);

        case1 = abs(b-c) < 1e-6; % singular Jacobian
        case2 = abs(sum(u)) < 1e-6; % undefined Jacobian

        if case1 & case2
            % very near the origin
            x = [-1;-1];
        elseif case1
            r = -(b*f1+a*f2)/(a*a+b*b);
            x = [r;r]/2;
        elseif case2
            x = [-f1;f1] / (b-c);

```

```

else
    % no special case; proceed as normal
    x = [c*f2-a*f1;a*f1-b*f2] / (a*(b-c));
end

%%%%% END NIFTY SINGULARITY SOLVING %%%%%

% the t-picker should return a t that satisfies:
%     norm(g(u+t*x))/norm(g(u)) < 1
t = pickt(u,x,normg);

% local min hack...
if t < 1e-6
    notdone = 0;
else

    % update all values for the next iteration
    u = u + t*x;
    k = k + 1;

    % Test the grid; maybe we've calculated an attraction point already?
    U = u(1,1)*ones(divs(2,1),divs(1,1));
    V = u(2,1)*ones(divs(2,1),divs(1,1));
    Tshift = abs(T(:, :, 1:2) - cat(3,U,V));
    % Find the four corners of the box we're in
    [ir,ic] = find(Tshift(:, :, 1) < (LRcorner(1,1)-ULcorner(1,1))/(divs(1,1)-1) & ...
                  Tshift(:, :, 2) < (ULcorner(2,1)-LRcorner(2,1))/(divs(1,1)-1));

    if max(size(ir)) > 0
        mismatch = 0;
        cornu = round(T(ir(1),ic(1),3)*1e5)*1e-5;
        cornv = round(T(ir(1),ic(1),4)*1e5)*1e-5;
        its = zeros(1,max(size(ir)));
        its(1,1) = T(ir(1),ic(1),5);
        if its(1,1) == -1
            % We haven't yet calculated an attraction point at this corner
            mismatch = 1;
        end
        % do all of the closest grid points attract to the same place?
        for ci=2:max(size(ir))
            its(1,ci) = T(ir(ci),ic(ci),5);
            if its(1,ci) == -1
                mismatch = 1;
            elseif cornu ~= round(T(ir(ci),ic(ci),3)*1e5)*1e-5 | ...
                   cornv ~= round(T(ir(ci),ic(ci),4)*1e5)*1e-5
                % we've had a disagreement
                mismatch = 1;
            end
        end
    end
end

```

```

        end
    end
    if mismatch == 0
        % all the closest gridpoints agree and have already been
        % calculated, so short-circuit the evaluation
        %disp('Woohoo! Shortcut!');
        notdone = 0;
        u = [cornu;cornv];
        k = k + round(median(its));
    end
end

gp = g(u);
normg = norm(gp);
end
%[u;x;t;normg]
end

%%%%% END NEWTON LOOP %%%%%

T(currpos(1,1),currpos(2,1),3) = u(1,1);
T(currpos(1,1),currpos(2,1),4) = u(2,1);
T(currpos(1,1),currpos(2,1),5) = k;
end

function [t] = pickt(u,x,normg)
% t/2 picker
t = 1;
un = u + t*x;
gn = g(un);
notdone = 1;
while (norm(gn)/normg > 1) & notdone
    if abs(t) < 1e-6
        % way too many iterations... we're at a local min.
        notdone = 0;
    else
        % keep halving t until it satisfies the exit condition
        t = t/2;
    end
    un = u + t*x;
    gn = g(un);
end

function [f] = g(u)
% initialize f as col vector
f=zeros(2,1);
f(1)= (u(1)-u(2)^7)^3 + u(2)^3;

```

```

sum = sum(u);
f(2) = sign(sum)*abs(sum)^.2 + 1;

function [J] = gprime(u)
J(1,1) = 3*(u(1)-u(2)^7)^2;
J(1,2) = (-7*u(2)^6)*J(1,1) + 3*u(2)^2;
sum = sum(u);
if sum==0
    J(2,:) = ones(1,2) * NaN;
else
    J(2,1) = .2/abs(sum)^.8;
    J(2,2) = J(2,1);
end

```

B. Part II. Matlab Code

```
% Constant-j Discretization ODE solver
```

```

function [arr] = constj_solver(beta_param, rel_trunc_tol)
    global beta
    tol_cutoff = 4e-3;
    % arr is a 3x(n+1) array when the interval is divided into n subintervals:
    % arr(1,i) is the horizontal position of the i-th interval division; arr(1,1) is
    %     0, the left endpoint, and arr(1,n+1) is 1, the right endpoint.
    % arr(2,i) is the estimated value of u at the i-th interval division; arr(2,1)
    %     and arr(2,n+1) are the boundary values.
    % arr(3,i) is the j-value for the interval *following* the i-th interval
    %     division; arr(3,n) is the j value for the last interval, and
    %     arr(3,n+1) is an undefined junk value.
    beta = beta_param;
    old = [0,1;0,0;0,0];
    old = calc_j(old);
    old = subdivide(old);
    drops = 0;
    while drops < size(old,2)-1
        [new,maxj] = subdivide(old);
        tol = maxj * rel_trunc_tol;
        if tol > tol_cutoff
            tol = tol_cutoff;
        end
        [new,drops] = pick_ints(old,new,tol);
        [drops,tol*1e4]
        old = new;
    end
    arr = new;

```

```

function [new,maxj] = subdivide(old)
    % old is the current 3x(n+1) [i;u;j] array
    % new will be the 3x(2n+1) [i;u;j] array for the next iteration
    % maxj will be the maximum, in absolute value, j-value found this iteration
    % Row 1: Halve the divisions
    new = zeros(3,2*size(old,2)-1);
    for i=1:size(old,2)
        new(1,2*i-1) = old(1,i);
    end
    for i=1:(size(old,2)-1)
        k = 2*i;
        new(1,k) = (new(1,k-1)+new(1,k+1))/2;
    end
    % Row 2: Fill in the u-values
    hv = h_vec(new);
    bv = b(new);
    [a,d,c] = construct_T(hv);
    [u] = solve_tri(a,d,c,bv');
    for i=1:size(u,1)
        new(2,i+1) = u(i,1);
    end
    % preserve the boundary values
    new(2,1) = old(2,1);
    new(2,size(new,2)) = old(2,size(old,2));
    % Row 3: Fill in the j-values
    [new,maxj] = calc_j(new);

function [a,d,c] = construct_T (h)
    % h should be a vector of h-values at this point in the problem
    s = size(h,2)-1;
    a = zeros(1,size(h,2)-1); d = zeros(1,size(h,2)-1); c = zeros(1,size(h,2)-1);
    for i=1:s
        if i > 1 % not the first row
            a(i-1) = du(h,i,-1);
        end
        d(i) = du(h,i,0) + (h(1,i)+h(1,i+1))/2;
        if i < s % not the last row
            c(i) = du(h,i,1);
        end
    end
end

function [y] = du(h,i,x)
    % x flag determines which calculation to do: du returns dj/d(u[i+x]).
    global beta
    if x ~= 1
        hleft = h(1,i);
        elleft = beta * hleft;
    end

```

```

end
if x ~= -1
    hright = h(1,i+1);
    elright = beta * hright;
end
if x == -1
    y = -(1/hleft)*(C(elleft) - elleft/2);
elseif x == 0
    y = (1/hright)*(C(elright) - elright/2) + (1/hleft)*(C(elleft) + elleft/2);
elseif x == 1
    y = -(1/hright)*(C(elright) + elright/2);
end

function [y] = B(x)
% Bernoulli function
% 5e-13 determined empirically as the smallest which Matlab handles correctly
if x < 5e-13
    y = 1;
else
    y = x / (exp(x) - 1);
end

function [y] = C(x)
y = B(abs(x)) + abs(x)/2;

function [h] = h_vec(arr)
% arr is a 3x(n+1) [i;u;j] array; h_vec creates a row vector of interval widths.
% The first and last elements of h are 0, the "external" intervals outside the
% boundary values.
for i=1:(size(arr,2)-1)
    h(1,i) = arr(1,i+1)-arr(1,i);
end

function [jv] = j_vec(arr)
% arr is a 3x(n+1) [i;u;j] array; j_vec creates a row vector of j-values for
% all the intervals.
jv = arr(3,1:(size(arr,2)-1));

function [arr,maxj] = calc_j(arr)
% arr is a 3x(n+1) [i;u;j] array with i and u values filled in;
% calc_j fills in the proper j values in the third row, and also
% returns the maximum j in absolute value.
% Note each j will be in the column corresponding to the u at the left
% end of its interval; therefore the lower-right corner of the matrix
% will not have a value.
jv = j_val(arr(2,1),arr(2,2),arr(1,2)-arr(1,1));
maxj = abs(jv);

```

```

arr(3,1) = jv;
for i=2:(size(arr,2)-1)
    % for each interval
    jv = j_val(arr(2,i),arr(2,i+1),arr(1,i+1)-arr(1,i));
    if abs(jv) > maxj
        maxj = abs(jv);
    end
    arr(3,i) = jv;
end

function [bv] = b(arr)
    % arr is a 3x(n+1) [i;u;j] array.
    % This function calculates the column vector b for use in the Tu=b calculation
    % for the positions indicated in the first row of arr. There are n divisions,
    % and therefore n+1 values of b.
    % NOTE: this function only requires the first row of arr to be defined.
    bv = zeros(size(arr,2)-2,1);
    for i=1:size(arr,2)-2
        x = arr(1,i+1);
        % b[i] = rhs(x[i]) * (h[left]+h[right])/2 = rhs(x[i]) * (x[i+1]-x[i-1])/2
        bv(i,1) = 100*exp(-64*(x-3/4)^2)*(arr(1,i+2)-arr(1,i))/2;
    end

function [next,d] = pick_ints(old, new, tol)
    % considering the previous iteration divided into n intervals
    % old is a 3x(n+1) [i;u;j] array from the previous iteration
    % new is a 3x(2n+1) [i;u;j] array from the current iteration
    % tol is the truncation tolerance; any truncation error > tol will
    %     preserve the subdivision
    % pick_ints returns an [i;u;j] array for use in the next iteration
    drop = [];
    for i=1:(size(old,2)-1)
        % for each old interval, calculate the subdivision's truncation error
        k = 2*i-1;
        truncerr = abs(2*old(3,i) - (new(3,k) + new(3,k+1)));
        if truncerr < tol
            % this interval passed the test, so drop it
            drop(size(drop,2)+1) = i;
            % recalculate j based on the new u estimates at the end of the large
            % (old) interval
            new(3,k) = j_val(new(2,k),new(2,k+2),new(1,k+2)-new(1,k));
            % new(3,k+1) will be ignored when building the final array; it is
            % the new j for the right subdivision, which is now being thrown away.
        end
    end
    % drop is now a row vector of indices in old corresponding to the left ends
    % of intervals that should not be subdivided.

```

```

next = zeros(3,size(new,2) - size(drop,2));
if size(drop,2) == 0
    % check for empty drop set to save some calculations
    next = new;
else
    nexti = 1;
    for i=1:(size(old,2)-1)
        if size(find(drop==i),2) > 0
            % drop this subdivision
            next(:,nexti) = new(:,(2*i-1));
            nexti = nexti + 1;
        else
            next(:,nexti:(nexti+1)) = new(:,(2*i-1):(2*i));
            nexti = nexti + 2;
        end
    end
    % now tack on the final u (and its undefined associated j)
    next(:,nexti) = new(:,size(new,2));
end
d = size(drop,2);

function [iv] = j_val(u1,u2,h)
    global beta
    l = beta*h;
    iv = C(1);
    iv = (iv+1/2)*u2 - (iv-1/2)*u1;
    iv = iv/h;

function [x] = solve_tri(a,d,c,b)
    % Solve tridiagonal matrix, assumed to be spd
    % All arguments are row vectors
    % a is the lower diagonal, d the main, c the upper
    % b is the rhs
    n = size(d,2);
    x = zeros(n,1);
    for i=2:n
        R = a(i-1)/d(i-1);
        d(i) = d(i) - R*c(i-1);
        b(i) = b(i) - R*b(i-1);
    end
    x(n) = b(n)/d(n);
    for i=n-1:-1:1
        x(i) = (b(i) - c(i)*x(i+1))/d(i);
    end
end

```