A Reduction Algorithm for Sublinear Elliptic Partial Differential Equations

by Sheldon H. Lee

A Thesis
Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in Mathematics

Northern Arizona University
December, 2000

Approved:

John M. Neuberger, Ph.D., Chair

Lawrence M. Perko, Ph.D.

James W. Swift, Ph.D.

Abstract

A Reduction Algorithm for Sublinear Elliptic Partial Differential Equations

Sheldon H. Lee

Our goal is to approximate solutions to the elliptic PDE:

$$\begin{cases} \Delta u + f(u) = 0 & \text{in } \Omega \\ u = 0 & \text{in } \partial\Omega, \end{cases}$$

where Ω is a region in $\mathbf{R}^N$. Solutions to this partial differential equation (PDE) are critical points of the functional $J : H \to \mathbf{R}$:

$$J(u) = \int_\Omega \{\frac{1}{2}|\nabla u|^2 - F(u)\}\, dx,$$

where $F(u) = \int_0^u f(x)\, dx$ and $H = H_0^{1,2}(\Omega)$.

Critical points of J are functions u such that $J'(u)(v) = 0$ for all $v \in H$. In our algorithms, we use a Fourier expansion of our function u. Our calculations are done to the Fourier approximation of u, as opposed to u itself.

We use three iterative algorithms to find three solutions: the one-sign solution, the minimal-engergy sign-changing solution, and a higher energy so-called reduction solution. Each algorithm makes use of a combination of steepest ascent and descent. The nonlinearity f is taken to be sublinear. Our algorithms use Sobolev gradients as opposed to L^2 gradients, since it is known that numerical approximations using the L^2 gradients behave poorly.

We will perform experiments which find approximate solutions, minimizing the residual and comparing our solution to solutions obtained by other methods. This thesis is based on [5], which was submitted to the proceedings of the World Congress of Nonlinear Analysts, 2000.

Acknowledgements

I would like to thank Dr. John Neuberger for providing me with an interesting and relevant topic. I would also like to thank him for taking several hours each week out of his busy schedule to assist me with this project. From him I learned not only a great deal about mathematics, but I learned how to go about conducting mathematical research. Dr. Neuberger has a big heart and infinite patience. Without him this thesis would not be possible.

I would also like to thank Dr. Lawrence Perko and Dr. James Swift for taking the time to read this thesis and providing me with useful feedback. A big thanks goes out to my family for their constant support throughout my educational career.

Contents

List of Figures

Chapter 1

Introduction

Let $\Omega \subset \mathbf{R}^N$ be a piecewise smooth bounded region and $f \in C^1(\mathbf{R}, \mathbf{R})$ be such that $f(0) = 0$. Let Δ be the Laplacian operator (see Appendix A.0.9). When $N = 2$ we take $\Omega = (0,1) \times (0,1)$. In this paper we study the nonlinear elliptic boundary value PDE

$$\begin{cases} \Delta u + f(u) = 0 & \text{in } \Omega \\ u = 0 & \text{in } \partial\Omega. \end{cases} \tag{1.1}$$

When $N = 1$ we take $\Omega = (0,1)$, whence the problem becomes the ODE

$$\begin{cases} u'' + f(u) = 0 & \text{in } (0,1) \\ u = 0 & \text{in } \{0,1\}. \end{cases} \tag{1.2}$$

We require f to be sublinear. That is, $f'(\infty) := \lim_{|u| \to \infty} f'(u) < \infty$. In particular, we use the following function:

$$f(x) = \begin{cases} ax + (b-a)\ln(1+x), & x \geq 0 \\ ax - (b-a)\ln(1-x), & x < 0. \end{cases} \tag{1.3}$$

Differentiating our function, we get

$$f'(x) = \begin{cases} a + \frac{b-a}{1+x}, & x \geq 0 \\ a + \frac{b-a}{1-x}, & x < 0. \end{cases}$$

One can see that $f(0) = 0$, $f'(\infty) = a$, and $f'(0) = b$. The advantage of this definition is that it is easy to change a and b, while preserving the function's

required properties. This is especially useful in constructing bifurification diagrams.

In order to find solutions to (1.1), we find critical points of the functional $J : H \to \mathbf{R}$ defined by

$$J(u) = \int_\Omega \{\frac{1}{2}|\nabla u|^2 - F(u)\}\, dx, \tag{1.4}$$

where $F(u) = \int_0^u f(x)\, dx$ and $H = H_0^{1,2}(\Omega)$. We will see that critical points of J will also be solutions to (1.1).

The eigenvalue problem in the case $N = 1$ is given as $\psi'' + \lambda\psi = 0$ on $(0, 1)$, $\psi(0) = \psi(1) = 0$. ODE theory tells us that the characteristic equation in this case will be $r^2 + \lambda = 0$, which gives $r = \pm i\sqrt{\lambda}$. Then, the solution will be $\psi = c_1 e^{i\sqrt{\lambda}x} + c_2 e^{-i\sqrt{\lambda}x}$. Because of the identity $e^{i\theta} = \sin(\theta) + i\cos(\theta)$, we have $\psi = c_1 \sin(\sqrt{\lambda}x) + c_1 i\cos(\sqrt{\lambda}x) - c_2 \sin(\sqrt{\lambda}x) + c_2 i\cos(\sqrt{\lambda}x)$. Since u takes on real values, we can simplify the above expression to $\psi = A\sin(\sqrt{\lambda}x) + B\cos(\sqrt{\lambda}x)$. Using the boundary condition $\psi(0) = 0$, we have $B = 0$. Applying $\psi(1) = 0$, we get $0 = A\sin(\sqrt{\lambda})$. Unless ψ is the trivial solution, $\sin\sqrt{\lambda} = 0$ is required. Thus, $\sqrt{\lambda} = k\pi$, or

$$\lambda_k = k^2\pi^2 \text{ for } k \in \mathbf{N}.$$

Our eigenfunction will be $\psi_k = A\sin(k\pi x)$. With the added requirement that ψ is normalized in $L^2 = L^2(\Omega, \mathbf{R})$ (see Appendix A.0.6), we will have:

$$\psi_k = \sqrt{2}\sin(k\pi x).$$

For the eigenvalue problem in the case $N = 2$, we consider $\Delta\psi + \lambda\psi = 0$ on $\Omega = (0, 1)\times(0, 1)$, $\psi = 0$ on $\delta\Omega$. Using separation of variables, we first assume that $\psi(x, y) = g(x)h(y)$. Then, $\Delta(gh) + \lambda gh = 0$, hence $g''h + gh'' + \lambda gh = 0$. This can be simplified to

$$\frac{g''}{g} + \frac{h''}{h} = -\lambda. \tag{1.5}$$

Then, since g and h are functions of different variables, $\frac{g''}{g}$ and $\frac{h''}{h}$ must both be constant. Solving $-\frac{g''}{g} = \alpha$ is the same problem as the ODE eigenvalue problem, so we get $g_j = \sin(j\pi x)$ and $\alpha_j = j^2\pi^2$. Similarily, $h_k = \sin(k\pi y)$ and $\beta_k = k^2\pi^2$. Substituting the values back into (1.5), we have $-(\alpha_j + \beta_k) = -\lambda$. Double indexing is natural here, so that

$$\lambda_{jk} = (j + k)^2\pi^2.$$

Using the fact that $\psi = gh$ and normalizing in $L^2(\Omega)$, we have

$$\psi_{jk} = 2\sin(j\pi x)\sin(k\pi y).$$

Using the Kronecker delta function ($\delta_{ij} = 0$ for $i \neq j$, $\delta_{ii} = 1$), we have that in the L^2 inner product (see Appendix A.0.6) we have

$$\langle \psi_i, \psi_j \rangle_2 = \int_\Omega \psi_i \psi_j \, dx = \delta_{ij}. \tag{1.6}$$

Thus, $\{\psi_j\}$ forms an orthonormal basis of L^2. We will see that $\{\psi_j\}$ forms an orthogonal basis for H, although it is no longer normal in that space.

The particular solution that we are most interested in is provided by the following theorem, which relies on the so-called Lyapunov-Schmidt reduction method. We call this solution the "reduction solution". For a complete proof of this theorem, refer to [2]. For the convenience of the reader, a proof of the key lemma used in the proof of the theorem is offered later in this chapter (see also [5]).

Theorem 1.0.1 *Let $X = \text{span}\{\psi_1, \cdots, \psi_k\}$, and $Y = X^\perp = \text{span}\{\psi_{k+1}, \cdots\}$. Let $f'(0) < \lambda_1, f'(\infty) \in (\lambda_k, \lambda_{k+1})$ with $k \geq 2$, and $f'(t) \leq \gamma < \lambda_{k+1}$. Let H be the real separable Hilbert space $H_0^{1,2}(\Omega)$. Let $J : H \to R$ be the functional defined in (1.4). Then there is a solution u such that $J(u) = \max_{x \in X} \min_{y \in Y} J(x + y)$.*

Note that in the above theorem, X and Y are closed subspaces of H such that $H = X \oplus Y$.

In our reduction algorithm (Algorithm B), we perform steepest ascent in the X components and steepest descent in the Y components. It turns out that this method will allow us to find $\max_{x \in X} \min_{y \in Y} J(x + y)$. See the remarks preceeding Algorithm B for more details on how this is accomplished. For the one-sign algorithm, in each iteration u is projected onto the set $S = \{u \in H - \{0\} : J'(u)(u) = 0\}$, after which a step in the $-\nabla J(u)$ direction is taken. For the sign-changing algorithm, u is instead projected onto $S_1 = \{u \in S : u_+ \in S, u_- \in S\}$, after which one follows $-\nabla J(u)$.

Theorem (1.0.1) follows directly from the following Lemma. We provide a sketch of the proof taken from [5], as it gives insight into the reduction algorithm.

Lemma 1.0.2 *Let H be a real separable Hilbert space. Let X and Y be closed subspaces of H such that $H = X \oplus Y$. Let $J : H \to \mathbf{R}$ be a functional of class C^1. If there exists $m > 0$ such that for all $x \in X$ and y, $y_1 \in Y$ we have*

$$\langle \nabla J(x+y) - \nabla J(x+y_1), y - y_1 \rangle \geq m\|y - y_1\|^2, \tag{1.7}$$

then the following hold:

(i) There exists a continuous function $\phi : X \to Y$ such that

$$J(x + \phi(x)) = \min_{y \in Y} J(x+y).$$

Moreover, $\phi(x)$ is the unique member of Y such that

$$\langle \nabla J(x + \phi(x)), y \rangle = 0 \qquad \text{for all} \quad y \in Y. \tag{1.8}$$

(ii) The function $\tilde{J} : X \to \mathbf{R}$ defined by $\tilde{J}(x) = J(x + \phi(x))$ is of class C^1, and

$$\langle \nabla \tilde{J}(x), x_1 \rangle = \langle \nabla J(x + \phi(x)), x_1 \rangle \qquad \text{for all} \quad x,\, x_1 \in X. \tag{1.9}$$

(iii) An element $x \in X$ is a critical point of $\tilde{J}$ if and only if $x + \phi(x)$ is a critical point of J.

(iv) If $-\tilde{J}$ is weakly lower semicontinuous and

$$J(x) \longrightarrow -\infty \qquad \text{as} \qquad \|x\| \to \infty \quad (x \in X) \tag{1.10}$$

Then there exists $u_0 \in H$ such that $\nabla J(u_0) = 0$ and

$$J(u_0) = \max_{x \in X} \min_{y \in Y} J(x+y).$$

It is later shown that our functional J satisfies all the hypotheses to this lemma with X and Y defined as in Theorem (1.0.1). Henceforth we refer to the solution u_0 as the "reduction solution". We now provide a sketch of the proof of Lemma 1.0.2.

For each $x \in X$ define $J_x : Y \to \mathbf{R}$ by $J_x(y) = J(x+y)$. Using condition (1.7) it is easy to show that J_x is weakly lower semicontinuous and coercive. Thus J_x has a unique minimum $\phi(x) \in Y$. Therefore,

$$J(x+\phi(x)) = \min_{y \in Y} J(x+y). \tag{1.11}$$

Because $J \in C^1(H, \mathbf{R})$, it follows that $J_x \in C^1(Y, \mathbf{R})$, and $\phi(x)$ is the only element of Y such that

$$0 = \langle \nabla J_x(\phi(x)), y \rangle = \langle \nabla J(x+\phi(x)), y \rangle \quad \forall y \in Y. \tag{1.12}$$

We now show that $\phi : X \to Y$ is a continuous function. Suppose ϕ is not continuous. Let $\delta > 0$ and $(x_n) \subset X$ such that

$$\lim_{n \to \infty} x_n = x \qquad \text{and} \qquad \|\phi(x_n) - \phi(x)\| \geq \delta.$$

Let P be the projection of H onto Y, and P^* be the adjoint of P. We observe that for any $x \in X$

$$P^* \nabla J(x+\phi(x)) = 0. \tag{1.13}$$

Using (1.13) and the continuity of ∇J and P^* we see that for n sufficiently large

$$\|P^* \nabla J(x_n + \phi(x))\| < m\delta. \tag{1.14}$$

From (1.7), (1.13), and the Cauchy-Schwarz inequality it follows that

$$\|P^* \nabla J(x_n + \phi(x))\| \geq m \, \|\phi(x_n) - \phi(x)\| \geq m \, \delta. \tag{1.15}$$

Inequality (1.15) contradicts (1.14). Thus, ϕ is continuous. This proves part (i).

Let $x, x_1 \in X$ and $t > 0$. Since ∇J and ϕ are continuous, using (1.11) we can see that

$$\lim_{t \to 0} \frac{\hat{J}(x + t\, x_1) - \hat{J}(x)}{t} = \langle \nabla J(x+\phi(x)), x_1 \rangle. \tag{1.16}$$

This shows that $\hat{J}$ has a continuous Gateaux derivative and hence is of class C^1. From above we have

$$\langle \nabla \hat{J}(x), x_1 \rangle = \langle \nabla J(x+\phi(x)), x_1 \rangle \ \ \forall x, x_1 \in X.$$

This proves part (ii).

Part (iii) follows from (1.8) and (1.9).

Since

$$-\hat{J}(x) = -J(x+\phi(x)) \geq -J(x) \quad \text{and } J(x) \to -\infty \qquad \text{as} \qquad \|x\| \to \infty,$$

it follows that

$$-\hat{J}(x) \longrightarrow +\infty \qquad \text{as} \qquad \|x\| \to \infty \quad (x \in X).$$

Therefore $-\hat{J}$ is weakly lower semicontinuous and coercive, and hence $-\hat{J}$ has a minimum. Consequently, there exists $x_0 \in X$ such that

$$\hat{J}(x_0) = \max_{x \in X} \hat{J}(x). \tag{1.17}$$

Since $\hat{J}(x) = J(x+\phi(x)) = \min_{y \in Y} J(x+y)$, we see that

$$J(x_0 + \phi(x_0)) = \max_{x \in X} \min_{y \in Y} J(x+y). \tag{1.18}$$

Also, since $\hat{J}$ is of class C^1, from (1.17) we have

$$\langle \nabla \hat{J}(x_0), x \rangle = 0 \quad \forall x \in X. \tag{1.19}$$

Let $x \in X$ and $y \in Y$.

$$\langle \nabla J(x_0 + \phi(x_0)), x+y \rangle = \langle \nabla J(x_0 + \phi(x_0)), x \rangle + \langle \nabla J(x_0 + \phi(x_0)), y \rangle \tag{1.20}$$

Using (1.8), (1.9), and (1.19) we see that the first term and the second term of the right hand side of (1.20) are equal zero. Thus if $u_0 = x_0 + \phi(x_0)$ we have $\nabla J(u_0) = 0$ and

$$J(u_0) = \max_{x \in X} \min_{y \in Y} J(x+y).$$

This proves part (iv), which concludes the proof of Lemma 1.0.2.

One can show that Lemma 1.0.2 does imply Theorem 1.0.1. Indeed,

$$\langle \nabla J(x+y) - \nabla J(x+y_1), y - y_1 \rangle_2 = J'(x+y)(y-y_1) - J'(x+y_1)(y-y_1)$$

$$= \langle x+y, y-y_1 \rangle_H - \textstyle\int_\Omega (y-y_1) f(x+y) - \langle x+y_1, y-y_1 \rangle_H - \int_\Omega (y-y_1) f(x+y_1)$$

$$= ||y-y_1||^2 - \textstyle\int_\Omega (y-y_1)^2 \frac{f(x+y)-f(x+y_1)}{(x+y)-(x+y_1)}\, dx = ||y-y_1||^2 - \int_\Omega f'(\xi)(y-y_1)^2,$$

where we have used the Mean Value Theorem. Now, $f'(\xi) \leq \gamma < \lambda_{k+1}$ and $y, y_1 \in Y$ imply that $y - y_1 \in Y$. Using a Poincare-Type Inequality,

$$||y - y_1||^2 \geq \lambda_{k+1} \int_\Omega (y - y_1)^2 \, dx,$$

so that

$$\langle \nabla J(x + y) - \nabla J(x + y_1), y - y_1 \rangle \geq (1 - \frac{\gamma}{\lambda_{k+1}}) ||y - y_1||^2 = m||y - y_1||^2.$$

The hypothesis that $f(0) = 0$ clearly implies that $u \equiv 0$ (trivial solution) satisfies 1.1 and 1.2. The trivial solution is a local minimum of J, and hence of Morse index (MI) zero. See Appendix A.0.12 for a brief discussion of Morse index. In previous works of Castro, Cossio, Neuberger, and others, it was proved that four additional solutions to (1.1) exist. Subsequent research investigated the properties of these four nontrivial solutions. In particular it was shown in [3], that if f is superlinear, there are three nontrivial solutions. Two of these three are of one sign and of MI one, while the third is a sign-changing exactly once MI two solution which we will call the "CCN" solution. These three solutions for the superlinear case were numerically computed in [7] using "Mountain Pass-Type Algorithms" derived from [3]. For an introduction to Mountain Pass-Type Theorems and Algorithms, see [10], [1], and [4]. In [9], Newton's Method along with Fourier expansions are used to find solutions of arbitrary MI. This algorithm does *not* follow from a constructive proof, unlike the three algorithms we present in this Thesis. In [9], the Newton Algorithm is applied to superlinear problems, although it can be applied to our sublinear case as well. In [2] an existence proof provides the fifth solution when $k \geq 2$. This fifth solution is of MI k and is the reduction solution provided by Theorem 1.0.1. This solution is the primary focus of this Thesis. We observe that if we allow $k = 1$, then the reduction solution coincides with either of the one-sign solutions.

Summarizing our methods, we will solve (1.1) using variational methods applied to the functional $J : H \to \mathbf{R}$. Critical points of J are solutions to (1.1). The reduction algorithm, which follows the proof of Theorem 1.0.1, is used to find critical points of J. We also use the two algorithms from [7] to find the positive, negative, and CCN solutions. These two algorithms are modified to also use Fourier approximations, as in [9] and in our new reduction algorithm.

Chapter 2

Numerical Considerations

Recall that our functional $J : H \to \mathbf{R}$ (see (1.4)) is defined by

$$J(u) = \int_\Omega \frac{1}{2}(\nabla(u))^2 - F(u)\,dx = \frac{1}{2}||u||_H^2 - \int_\Omega F(u)\,dx.$$

Under our hypothesis on f, one can show that $J \in C^2$ (see [1]). We can find $J'(u)(v)$ by computing its directional deriviative

$$\begin{aligned} J'(u)(v) &= \lim_{t\to 0} \tfrac{J(u+tv)-J(u)}{t} \\ &= \lim_{t\to 0} \tfrac{1}{t}\int_\Omega \{\tfrac{\nabla(u+tv)\cdot\nabla(u+tv)}{2} - F(u+tv) - \tfrac{\nabla u\cdot\nabla u}{2} + F(u)\}\,dx \\ &= \lim_{t\to 0} \int_\Omega \{\nabla u\cdot\nabla v + \tfrac{t}{2}|\nabla v|^2 - \tfrac{(F(u+tv)-F(u))v}{tv}\}\,dx \\ &= \int_\Omega \{\nabla u\cdot\nabla v - f(u)v\}\,dx, \end{aligned}$$

where we have applied the Lebesgue Dominated Convergence Theorem to take the limit inside the integral and used the fact that $F' = f$. Thus,

$$J'(u)(v) = \langle u, v\rangle_H - \int_\Omega v f(u)\,dx. \tag{2.1}$$

We now have a way of expressing $J'(u)(v)$ in general. However, in our algorithms, we only require calculating $J'(u)(\psi_j)$. Therefore, using our Fourier coefficients, we can rewrite (2.1) in a way which will make our numerical calculations simpler by avoiding some of the numerical differentiation and

integration. First we observe that

$$\begin{aligned} \langle u, v\rangle_H &= \int_\Omega \nabla u \cdot \nabla v \, dx = \int_{\partial\Omega} \frac{\partial u}{\partial \eta} v \, dx - \int_\Omega \Delta u v \, dx \\ &= \int_\Omega (-\Delta u) v \, dx = \langle -\Delta u, v\rangle_2. \end{aligned} \tag{2.2}$$

Note that we have used the fact that $v \equiv 0$ on the boundary $\partial\Omega$, so that when integrating by parts the term involving the outward unit normal η vanishes. Now using the fact that ψ_i is an eigenfunction of $-\Delta$ and that $\{\psi_i\}$ is orthonormal in L^2 (see (1.6)),

$$\langle \psi_i, \psi_j\rangle_H = \langle -\Delta\psi_i, \psi_j\rangle_2 = \langle \lambda_i\psi_i, \psi_j\rangle_2 = \lambda_i\delta_{ij} \tag{2.3}$$

We can now simplify $J'(u)(\psi_j)$. Recall that $J'(u)(\psi_j) = \langle u, \psi_j\rangle_H - \int_\Omega \psi_j f(u)\, dx$ (see (2.1)). Using the Fourier expansion $u = \sum_{i=1}^\infty a_i\psi_i$,

$$\begin{aligned} \langle u, \psi_j\rangle_H &= \langle \textstyle\sum_{i=1}^\infty a_i\psi_i, \psi_j\rangle_H = \sum_{i=1}^\infty a_i \langle \psi_i, \psi_j\rangle_H \\ &= \textstyle\sum_{i=1}^\infty a_i\lambda_i\delta_{ij} = a_j\lambda_j. \end{aligned}$$

Thus,

$$J'(u)(\psi_j) = a_j\lambda_j - \int_\Omega \psi_j f(u)\, dx.$$

Note that we have used (2.3), and the fact that every term of $\sum_{i=1}^\infty a_i\lambda_i\delta_{ij}$ will be zero except when $i = j$. The advantage of this expression is that we are not required to calculate $\int_\Omega \nabla u \cdot \nabla\psi_j \, dx$ numerically.

In order to show that critical points of J are solutions to (1.1), we must assume a regularity result (see [3] or [6]). This result states that if $\nabla J(u) = 0$ then we can conclude that $u \in C^2$. We can then show that critical points of J are solutions to the PDE. Recall from (2.2) that $\langle u, v\rangle_H = -\int_\Omega \Delta u v \, dx$, which gives us

$$J'(u)(v) = \langle u, v\rangle_H - \int_\Omega v f(u)\, dx = -\int_\Omega v(\Delta u + f(u))\, dx.$$

If u is a critical point, then $J'(u)(v) = 0$ for all $v \in H$. This forces $\Delta u + f(u) = 0$, therefore u will be a solution to the PDE.

We wish to now express $J(u)$ using Fourier coefficients. First we observe that

$$\begin{aligned} ||u||_H^2 &= \langle u, u\rangle_H = \langle \textstyle\sum_{i=1}^\infty a_i\psi_i, \sum_{j=1}^\infty a_j\psi_j\rangle_H \\ &= \textstyle\sum_{i=1}^\infty \sum_{j=1}^\infty a_i a_j \langle \psi_i, \psi_j\rangle_H = \sum_{i=1}^\infty \sum_{j=1}^\infty a_i a_j \lambda_i \delta_{ij}. \\ &= \textstyle\sum_{i=1}^\infty a_i^2 \lambda_i. \end{aligned}$$

Note that we have used (2.3) in the preceeding steps, as well as the fact that every term of $\sum_{i=1}^{\infty}\sum_{j=1}^{\infty} a_i a_j \lambda_i \delta_{ij}$ will equal zero unless $i = j$. We can now express $J(u)$ as

$$J(u) = \frac{1}{2}\sum_{i=1}^{\infty} a_i^2 \lambda_i - \int_{\Omega} F(u)\, dx.$$

When finding the residuals $||\Delta u + f(u)||_2$ numerically, it is helpful to use the Fourier coefficients in order to find Δu in order to avoid performing tedious divided differences in our code. Again using the fact that $\Delta\psi_i = -\psi_i\lambda_i$,

$$\Delta u = \Delta\sum_{i=1}^{\infty} a_i\psi_i = \sum_{i=1}^{\infty} a_i \Delta\psi_i = \sum_{i=1}^{\infty} a_i(-\psi_i\lambda_i) = -\sum_{i=1}^{\infty} a_i\lambda_i\psi_i.$$

In our algorithms, we will use the Sobolev gradient $\nabla_H J(u)$. The L^2 gradient $\nabla_2 J(u)$ is only densely defined. Not surprisingly, numerical approximations of $\nabla_2 J(u)$ behave poorly (see [8]). If $u \in C^2$, then $\nabla_2 J(u)$ is defined and $\nabla_2 J(u) = \sum_{i=1}^{\infty} J'(u)(\psi_i)\psi_i$.

Using the fact that $J'(u)(v) = \langle \nabla_2 J(u), v\rangle_2 = \langle \nabla_H J(u), v\rangle_H$ by definition, and the fact that $\langle u, v\rangle_H = \langle -\Delta u, v\rangle_2$ (see (2.2)) in turn, we have that

$$\langle \nabla_2 J(u), v\rangle_2 = \langle \nabla_H J(u), v\rangle_H = \langle -\Delta(\nabla_H J(u)), v\rangle_2.$$

This expression requires that the first component of each L^2 inner product equal each other, so that $\nabla_2 J(u) = -\Delta(\nabla_H J(u))$. Simplifying $\nabla_H J(u)$, we get

$$\nabla_H J(u) = -\Delta^{-1}(\nabla_2 J(u)) = -\Delta^{-1}\sum_{i=1}^{\infty} J'(u)(\psi_i)\psi_i = \sum_{i=1}^{\infty} J'(u)(\psi_i)(-\Delta^{-1}(\psi_i)).$$

Since $-\Delta\psi_i = \lambda_i\psi_i$, we have $-\Delta^{-1}(\psi_i) = \frac{\psi_i}{\lambda_i}$. Hence,

$$\nabla_H J(u) = \sum_{i=1}^{\infty} J'(u)(\psi_i)\frac{\psi_i}{\lambda_i}. \tag{2.4}$$

We have found that to find the Sobolev gradient, one only has to divide each i^{th} component of the L^2 gradient by λ_i.

Chapter 3

The Algorithms

For the one-sign algorithm, in each iteration u is projected onto the codimension one submanifold of H (see for example [3]), $S = \{u \in H - \{0\} : J'(u)(u) = 0\}$, after which one takes a step in the $-\nabla J(u)$ direction. For the sign-changing "CCN" algorithm, u is projected onto $S_1 = \{u \in S : u_+ \in S, u_- \in S\}$, after which one follows $-\nabla J(u)$. The reduction algorithm is similar in nature to Newton's Method, with steepest ascent in the X directions and steepest descent in the Y directions, where $X = \text{span}\{\psi_1, \psi_2, ..., \psi_k\}$, and $Y = \text{span}\{\psi_{k+1}, \psi_{k+2}, ...\}$.

For the following algorithms, M is the number of basis elements, so that our approximating subspace is $G = \text{span}\{\psi_1, \psi_2, ..., \psi_M\} \approx X \oplus Y = H$. In the ODE case the singly indexed basis has size $\hat{M} = M$, whereas for convenience we refer to the size of the doubly indexed basis for the PDE when $\Omega = (0,1) \times (0,1)$ as $\hat{M} = \sqrt{M}$. The numerical integration is accomplished by treating u as an array of values over a suitable grid on Ω. We use T divisions, and understand that there are $T+1$ grid points in the ODE case and $(T+1)^2$ grid points in the PDE case. Algorithm A1 and A2 are essentially as in [7], with the only difference being the use of Fourier approximations.

In [3], a theorem guarantees the existence of three nontrivial solutions to a class of superlinear problems (where $f'(\infty) = \infty$). In particular, there exists a pair of one-sign (positive and respectively negative) solutions which are local minimums of $J|_S$ and a global minimum of $J|_{S_1}$ which changes sign exactly once. The minimal energy sign changing solution is the one we have called the CCN solution. These solutions persist in our sublinear case, given some additional assumptions. In particular, the one-sign solutions exist if $f'(\infty) > \lambda_1$ and the CCN solution exists if $f'(\infty) > \lambda_2$.

The single constraint of membership in S implies that (if nondegenerate) the one-sign minimizers are of MI one, while the two constraints of membership in S_1 imply that the CCN solution is of MI two.

To find these minimizers numerically (see [7]), one projects iteratates onto S (Algortithm A1) or S_1 (Algorithm A2) and performs steepest descent by taking a step in the search direction $-\nabla J(u)$. As previously mentioned, we use the Sobolev gradient. Algorithms A1 and A2 differ from those found in [7] only in that the Fourier expansion approach of [9] is used.

Algorithm A1 (the one-sign algorithm).

Choose a function f (typically sublinear or superlinear) and stepsizes δ_1 and δ_2.
Choose $a = a^0 \in \mathbf{R}^M$ to be initial Fourier coefficients.
Set $u = u^0 = \sum_{i=1}^M a_i \psi_i$.
Loop counter $n = 0$
    Loop counter $m = 0$ (to project u onto S)
        Calculate $t = \frac{\sum_{i=1}^M a_i^2 \lambda_i - \int_\Omega u f(u)}{\sum_{i=1}^M a_i^2 \lambda_i}$, so that $P_u \nabla J(u) = tu$.
        Set $a = a^{m+1} = a^m + \delta_1 t a^m$ (steepest ascent in ray direction).
        Increment m.
        If $|a^{m+1} - a^m|$ is small, exit loop.
    Set $g = g^n = \{J'(u)(\psi_i)\}_{i=1,\dots M} \in \mathbf{R}^M$, so that $P_G \nabla_2 J(u) = \sum_{i=1}^M g_i \psi_i$.
    For $i = 1$ to M (to take a step in the $-\nabla J(u)$ direction)
        $a_i = a_i^{n+1} = a_i^n - \frac{\delta_2}{\lambda_i} g_i^n$.
    Set $u = u^{n+1} = \sum_{i=1}^M a_i \psi_i$ (Fourier expansion).
    Increment n.
    If $|g| \approx ||\nabla_2 J(u)||_2$ is small, exit loop.

Algorithm A2 (the CCN algorithm).

Choose a function f (typically sublinear or superlinear) and stepsizes δ_1 and δ_2.
Choose $a = a^0 \in \mathbf{R}^M$ to be initial Fourier coefficients.
Set $u = u^0 = \sum_{i=1}^M a_i \psi_i$.
Loop counter $n = 0$
    Calculate u_+, u_-, then a_+, a_- ($a_i = \langle u, \psi_i \rangle$).
    Loop counter $m = 0$ (to project u_+ onto S)
        Calculate $t = \frac{\sum_{i=1}^M a_{+i}^2 \lambda_i - \int_\Omega u_+ f(u_+)}{\sum_{i=1}^M a_{+i}^2 \lambda_i}$, so that $P_{u+} \nabla J(u_+) = tu_+$.

Set $a_+ = a_+^{m+1} = a_+^m + \delta_1 t a_+^m$ (steepest ascent in ray direction).
Increment m.
If $|a_+^{m+1} - a_+^m|$ is small, exit loop.
Loop counter $m = 0$ (to project u_- onto S)
Calculate $t = \frac{\sum_{i=1}^M a_{-i}^2 \lambda_i - \int_\Omega u_- f(u_-)}{\sum_{i=1}^M a_{-i}^2 \lambda_i}$, so that $P_{u_-} \nabla J(u_-) = t u_-$.
Set $a_- = a_-^{m+1} = a_-^m + \delta_1 t a_-^m$ (steepest ascent in ray direction).
Increment m.
If $|a_-^{m+1} - a_-^m|$ is small, exit loop.
Set $a = a_+ + a_-$.
Set $u = u^{n+1} = \sum_{i=1}^M a_i \psi_i$ (Fourier expansion).
Set $g = g^n = \{J'(u)(\psi_i)\}_{i=1,\dots M} \in \mathbf{R}^M$, so that $P_G \nabla_2 J(u) = \sum_{i=1}^M g_i \psi_i$.
For $i = 1$ to M (to take a step in the $-\nabla J(u)$ direction)
$a_i = a_i^{n+1} = a_i^n - \frac{\delta_2}{\lambda_i} g_i^n$.
Set $u = u^{n+1} = \sum_{i=1}^M a_i \psi_i$ (Fourier expansion).
Increment n.
If $|g| \approx ||\nabla_2 J(u)||_2$ is small, exit loop.

In Lemma 1.0.2, it was shown that there exists a function $\phi : X \to Y$ such that $J(x + \phi(x)) = \min_{y \in Y} J(x + y)$. It seems that for each iteration, one would have to first find $\phi(x)$ to minimize in Y, then take a step in the $+\nabla J(u)$ direction to maximize in X. We will actually take a slightly different approach. In our algorithm, the vector g is the first M components of the the L^2 gradient $\nabla_2 J(u)$. We develop a new vector $\tilde{g}$ which has two goals. First we require $\tilde{g}$ to represent the H gradient $\nabla_H J(u)$, so we take $\tilde{g}_i = \frac{g_i}{\lambda_i}$ for $i = 1, 2, \cdots, M$ (see 2.4.) Secondly, $\tilde{g}$ is taken to be negative in the Y components, and we therefore take $\tilde{g}_i = -g_i$ for $i = k+1, k+2, \cdots$. This new vector $\tilde{g}$ is what we call the "search direction". In each iteration of our algorithm, we will take a step in the $\tilde{g}$ direction.

In general terms, one could say that we take two steps in each iteration. We perform steepest ascent in the X components by taking a step in the $+\nabla J(u)$ direction. In the Y components we perform steepest descent by taking a step in the $-\nabla J(u)$ direction. As far as we are aware of, this "minimax" technique is original to this thesis.

Algorithm B (the reduction algorithm).

Choose a function f which is sublinear and stepsize δ.

Let k be the crossing eigenvalue number (i.e., $f'(\infty) \in (\lambda_k, \lambda_{k+1})$).
Choose $a = a^0 \in R^M$ to be initial Fourier coefficients.
Set $u = u^0 = \sum_{i=1}^{M} a_i \psi_i$.
Loop counter $n = 0$

$g = g^n = \{J'(u)(\psi_i)\}_{i=1,\ldots M} \in \mathbf{R}^M$, so that $P_G \nabla_2 J(u) = \sum_{i=1}^{M} g_i \psi_i$.
For $i = 1$ to k

$\tilde{g}_i = -\frac{1}{\lambda_i} g_i$.
For $i = k+1$ to M

$\tilde{g}_i = +\frac{1}{\lambda_i} g_i$.
Set $a = a^{n+1} = a^n - \delta \tilde{g}^n$.
Set $u = u^{n+1} = \sum_{i=1}^{M} a_i \psi_i$.
Increment n.
If $|g| \approx ||\nabla_2 J(u)||_2$ is small, exit loop.

In all of our included experimental results, we used step sizes $\delta = \delta_1 = \delta_2 = 0.1$, although larger step sizes could have been used. Algorithm A2 is very similar to Algorithm A. The main difference is that instead of projecting u onto S, one projects u onto S_1, where $P_{S_1} u = P_S u_+ + P_S u_-$. We found that our results in running this algorithm were not as good as in [7]. This could be due to trying to estimate a function such as $\{\sin 2\pi x\}_+$ using a Fourier expansion. We are not including any results from these experiments, with the exception of Figure 5.6.

We implemented code for algorithms A1 and B using C++. Code segments for the ODE case of Algorithm A1 are provided in Appendix D. In addition, code segments for both the ODE and PDE cases of Algorithm B are provided in Appendices B and C, respectively.

Chapter 4

ODE Results

In this section we will use Algorithms A1 and B to solve the problem

$$u'' + f(u) = 0 \text{ in } (0,1)$$

$$u(0) = u(1) = 0$$

For all experiments, unless otherwise stated we use the function

$$f(x) = \begin{cases} ax + b\ln(1+x) - a\ln(1+x) & x \geq 0 \\ ax - b\ln(1-x) + a\ln(1-x) & x < 0, \end{cases} \tag{4.1}$$

where $f'(\infty) = a$ and $f'(0) = b$. We performed all numerical integration using the Trapezoid Rule, although certainly more sophisticated quadrature methods could be used. Unless otherwise noted, the algorithms stop when $||\nabla_H J(u)||_2 < 10^{-6}$. We set $b = 0$ for all experiments, unless otherwise noted. For the tables shown in this section, we used $T = \hat{M}$.

Using Algorithm B, we numerically computed the solutions where $f'(\infty) \in (\lambda_1, \lambda_2)$ (one-hump solutions) and the case where $f'(\infty) \in (\lambda_2, \lambda_3)$ (two-hump solutions). In particular, for k = 1, $a = 2.5\pi^2$ and for k = 2, $a = 6.5\pi^2$. For Algorithm B in Table 4, we stopped execution when $|g| \approx ||\nabla_2 J(u)||_2 < 10^{-8}$ and compared the solutions to the Shooting Method (see Appendix E) solutions. Table 4.2 shows the computed residuals for Algorithm B, $k = 1$ and $k = 2$, as well as for Algorithm A1.

Bifurification diagrams were done which examine the relationship between $f'(\infty), f'(0)$ and $||u||_\infty$. The diagrams provided correspond with Algorithm B, $k = 1, 2, 3$. In the case for $k = 1$, the bifurification curve from the

Algorithm A1 almost exactly overlaps the curve obtained by Algorithm B, so we can omit bifurification diagrams from Algorithm A1.

In the case where we hold $f'(0)$ constant and vary $f'(\infty)$ (see Figure 4.1), recall that $f'(\infty) \in (\lambda_k, \lambda_{k+1})$ is required by Theorem 1.0.1. When $f'(\infty) > \lambda_{k+1}$, the dashed line represents the continuation of the reduction solution branch, although these "solutions" are no longer reduction solutions and cannot be stable under our algorithm. In addition, the graph has a vertical asymptote at $f'(\infty) = \lambda_k$. Note that $\lambda_1 \approx 9.9, \lambda_2 \approx 39.5$, and $\lambda_3 \approx 157.9$.

In addition, we plotted several of the solutions in the ODE case. The graphs of the solutions for Algorithm B, for $k = 1$ and $k = 2$ are shown. In Figure 4.3, where $k = 1$, solutions obtained from both Algorithm A1 and Algorithm B are both graphed, and are found to coincide.

Table 4.1: Approximation of u. The number of decimal places show agreement of the Algorithm B approximation with that generated by the Shooting Method.

x	.1	.3	.5	.7	.9
$T = \hat{M} = 1000, k = 1$	0.55730	1.5241	1.92140	1.5241	0.55730
$T = \hat{M} = 1000, k = 2$	3.074	5.091	-10^{-8}	-5.091	-3.074

Table 4.2: $||u'' + f(u)||_2$ with $T = \hat{M}$.

T	10	50	100	1000
Reduction, $k = 1$	0.1985	0.009554	0.002446	0.00002597
Reduction, $k = 2$	5.7395	0.3834	0.1079	0.001207
One-Sign	0.2013	0.01227	0.005741	0.004038

Figure 4.1: $||u||_\infty$ versus $f'(\infty), k = 1, 2, 3$

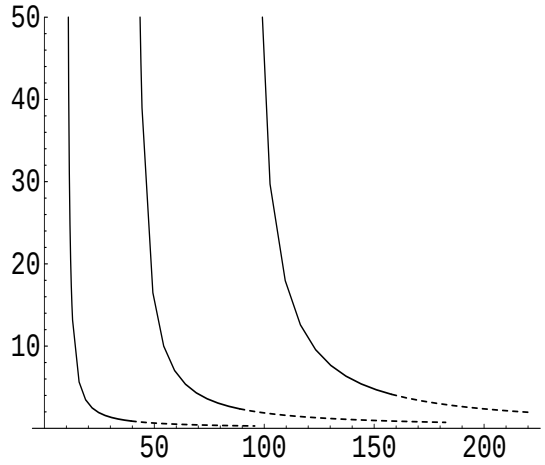


Figure 4.2: $||u||_\infty$ versus $f'(0), k = 1, 2, 3$

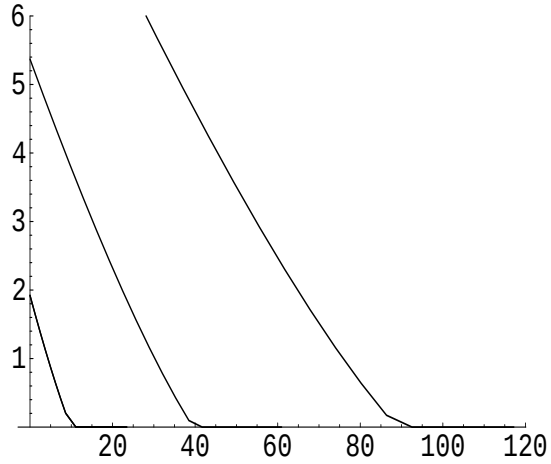


Figure 4.3: $u, k = 1$

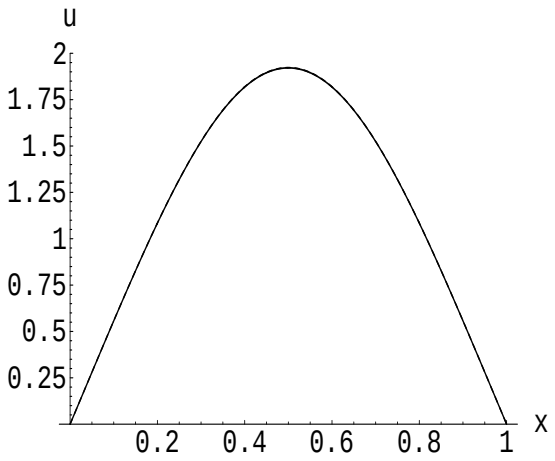
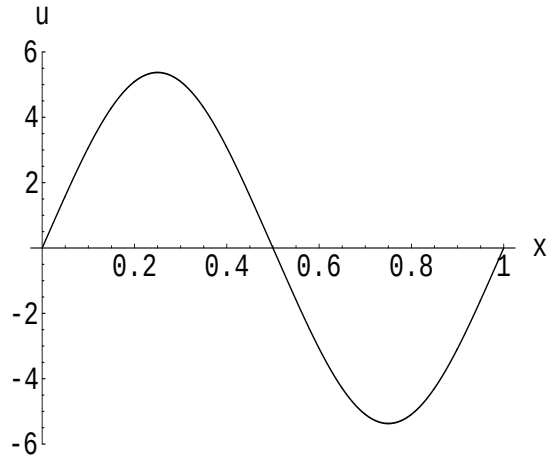


Figure 4.4: $u, k = 2$



Chapter 5

PDE Results

It should be noted that for the PDE, $||\Delta u + f(u)||_2$ was calculated using $\Delta u \approx \sum_{i=1}^{M} a_i \lambda_i \psi_i$, whereas in the ODE algorithms, divided differences were used. The PDE results for both Algorithms A1 and B were good. After a certain point, however, increasing n and m had no significant effect on the residual $||\Delta u + f(u)||_2$.

Table 5.1: u, Reduction Algorithm B, $k = 3$

	0.1	0.3	0.5	0.7	0.9
0.1	2.557	7.446	9.589	7.446	2.557
0.3	4.23	12.27	15.77	12.27	4.23
0.5	-10^{-9}	-10^{-8}	-10^{-8}	-10^{-8}	-10^{-9}
0.7	-4.23	-12.27	-15.77	-12.27	-4.23
0.9	-2.557	-7.446	-9.589	-7.446	-2.557

Table 5 shows that in our code's execution, the optimal relationship between T and $\hat{M}$ is roughly $T = \hat{M}$. We therefore used $T = \hat{M}$ in Table 5. We are at a loss to explain why refining the grid (increasing T) and keeping $\hat{M}$ constant does not result in a reduced residual.

In Figure 5, we demonstrate the presence of a solution which is unstable with respect to Algorithm B. The function u_R is an estimate of the reduction solution, obtained using $u_0 = \psi_{12}$ and stopping when $||\nabla_2 J(u)||_2 < 10^{-6}$. Using the initial guess $u_0 = \psi_{12} + \psi_{21}$ and stopping when $||\nabla_2 J(u)||_2 < 10^{-6}$, we saved off $u = u_C$, an estimate of the CCN solution. Again executing the algorithm with $u_0 = \psi_{12} + \psi_{21}$, we generate Figure 5. Since the CCN

Table 5.2: $||\Delta u + f(u)||_2$, Reduction Algorithm B, $k = 1$

$\hat{M} \backslash T$	$T = 10$	$T = 20$	$T = 50$
$\hat{M} = 10$	0.0000198	0.347	0.408
$\hat{M} = 20$	147	0.0000194	0.0863
$\hat{M} = 50$	1040	401	0.0000194

Table 5.3: Convergence data for the PDE Reduction Algorithm B

k	$T = \hat{M}$	$\|\|\Delta u + f(u)\|\|_2$	$J(u)$
1	10	0.0000198	11.6
1	50	0.0000194	11.6
3	10	0.0000479	168
3	50	0.0000482	172

solution is of Morse index 2 and is not stable, in time the algorithm converges to the Morse index 3 solution u_R, although it first "loiters" near u_C. A similar experiment was done in [7], where the sign-changing algorithm was run with $u_0 = \psi_{12}$, to show that the algorithm will eventually converge to the CCN solution, after loitering near u_R, which is unstable with respect to that algorithm.

Bifurification diagrams were done which examine the relationship between $f'(\infty), f'(0)$ and $||u||_\infty$, as in the ODE case. Again, if we hold $f'(0)$ constant and change $f'(\infty)$, (see Figure 5.2), we get the trivial solution when $f'(\infty) \geq \lambda_{k+1}$, and we indicate these portions of the graph with dashed lines. In both diagrams, bifurification curves are provided for Algorithm B, $k = 1, 3$. In the case for $k = 1$, bifurification curves from Algorithms A1 and B overlap, so we omit the Algorithm A1 bifurification diagram.

In addition, we plotted several of the reduction solutions from Algorithm B in the PDE case. In the case for $k = 1$, this solution is the same as in Algorithm A1. Recall that the solution for $k = 3$ will not be the same as in Algorithm A2 (see Figure 5), since Algorithm B gives a Morse index 3 solution, but Algorithm A2 gives a sign-changing solution of Morse index 2. This sign-changing solution is given in Figure 5.6.

Figure 5.1: Reduction algorithm, run with the "CCN" initial guess of $\psi_{12} + \psi_{21}$. The dashed line represents $||u - u_C||_2$, while the solid line represents $||u - u_R||_2$.

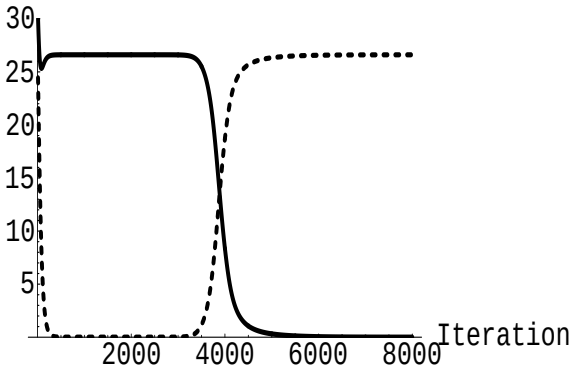


Figure 5.2: $||u||_\infty$ vs. $f'(\infty), k = 1, 2$

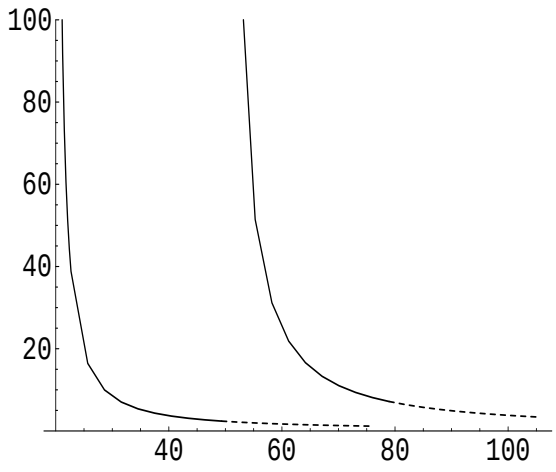


Figure 5.3: $||u||_\infty$ vs. $f'(0), k = 1, 2$

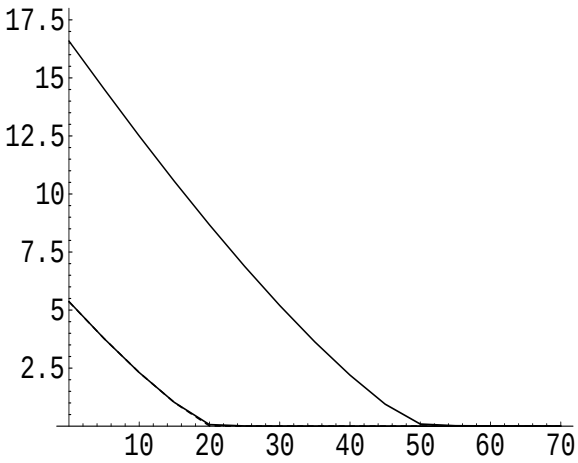


Figure 5.4: Reduction Solution, $k = 1$

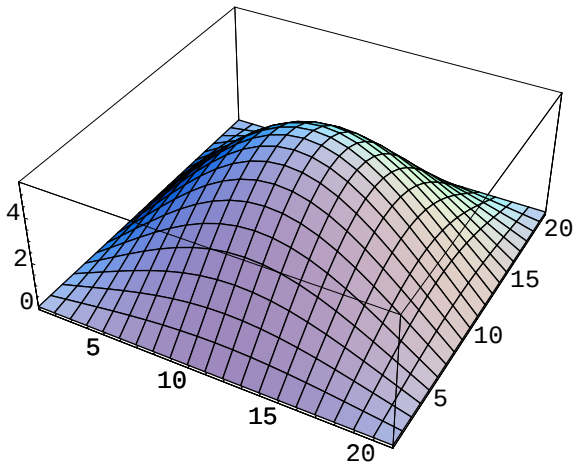


Figure 5.5: Reduction Solution, $k = 3$

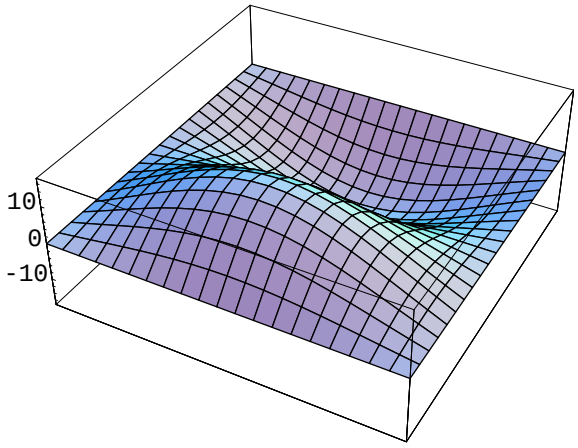


Figure 5.6: CCN solution

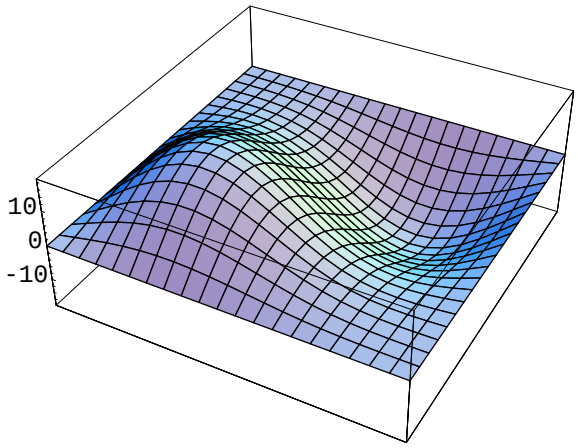
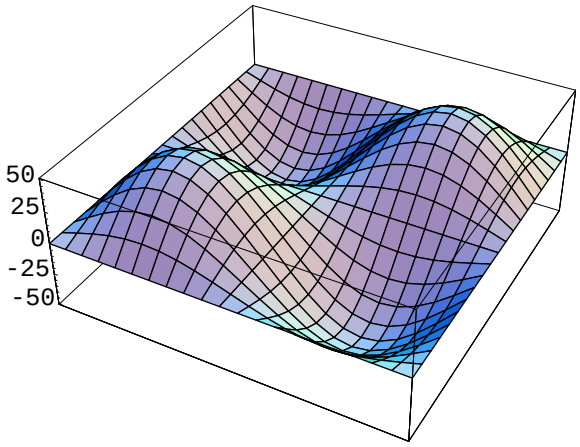


Figure 5.7: $u, k = 4$



Bibliography

[1] A. Ambrosetti and P. Rabinowitz, *Dual Variational Methods in Critical Point Theory and Applications*, Journal of Functional Analysis 14 (1973), 349-381.

[2] A. Castro and J. Cossio, *Multiple solutions for a nonlinear Dirichlet problem*, SIAM J. Math. Anal. **25** (1994), no. 6, p1554–1561.

[3] A. Castro, J. Cossio and J. M. Neuberger, *A Sign-Changing Solution for a Superlinear Dirichlet Problem*, Rocky Mountain J. of Math., **27**, No. 4 (1997), pp. 1041-1053.

[4] Y.S. Choi and P.J. McKenna, *A Mountain Pass Method for the Numerical Solutions of Semilinear Elliptic Problems*, Nonlinear Analysis, **20** (1993), p417-437.

[5] J. Cossio, S. Lee, and J. M. Neuberger, *A Reduction Algorithm for Sublinear Dirichlet Problems*, Accpeted by Nonlinear Analysis, WCNA 2000 proceedings, (2000).

[6] D. Gilbarg and N. Trudinger, *Elliptic Partial Differential Equations of Second Order*, Berlin, New York: Springer-Verlag, (1983).

[7] John M. Neuberger, *A Numerical Method for Finding Sign-Changing Solutions of Superlinear Dirichlet Problems*, Nonlinear World 4 (1997), no. 1, 73–83.

[8] J.W.Neuberger, *Sobolev Gradients and Differential Equations*, Spring Lecture Notes, (1997).

[9] John M. Neuberger and James W. Swift, *Newton's method and Morse index for semilinear elliptic PDE*, Accepted by Int. J. Bif. Chaos, (2000).

[10] P. Rabinowitz, *Minimax Methods in Critical Point Theory with Applications to Differential Equations*, Regional Conference Series in Mathematics, **65**, Providence, R.I.: AMS, (1986).

Appendix A

Background

Definition A.0.3 *Let V be a vector space over $\mathbf{R}$. An inner product on V is a function $\langle \cdot, \cdot \rangle : V \times V \to \mathbf{R}$ satisfying:*

- $\langle \alpha v, w \rangle = \alpha \langle v, w \rangle$ *for all* $\alpha \in \mathbf{R}, \mathbf{v}, \mathbf{w} \in \mathbf{V}$.
- $\langle v + w, u \rangle = \langle v, u \rangle + \langle w, u \rangle$ *for all* $v, w, u \in V$.
- $\langle v, w \rangle = \langle w, v \rangle$ *for all* $v, w \in V$.
- $\langle v, v \rangle \geq 0$ *for all* $v \in V$ *and if* $v \neq 0, \langle v, v \rangle > 0$.

We call V an inner product space.

Definition A.0.4 *Let V be a vector space over $\mathbf{R}$ and let $|| \cdot || : V \to [0, \infty)$. Then $|| \cdot ||$ is a norm on V if*

- $||v|| \geq 0$ *for all* $v \in V$ *and* $||v|| > 0$ *if* $v \neq 0$.
- $||\alpha v|| = |\alpha| ||v||$ *for all* $v \in V, \alpha \in \mathbf{R}$.
- $||v + w|| \leq ||v|| + ||w||$ *for all* $v, w \in V$.

Definition A.0.5 *A Hilbert Space is a inner product space in which every Cauchy sequence converges.*

Definition A.0.6 *The L^2 space is a Hilbert space with inner product*

$$\langle u, v \rangle_2 = \int_\Omega uv \, dx.$$

and norm

$$||u||_2 = (\int_\Omega u^2 \, dx)^{\frac{1}{2}} < \infty.$$

Definition A.0.7 *The Sobolev space $H = H_0^{1,2}(\Omega)$ is a Hilbert space and is a dense subspace of $L^2(\Omega)$, with inner product*

$$\langle u, v\rangle_H = \int_\Omega \nabla u \cdot \nabla v \, dx.$$

and norm

$$||u||_H = (\int_\Omega |\nabla u|^2 \, dx)^{\frac{1}{2}} < \infty.$$

The space $C_0^1(\bar{\Omega})$, the set of all functions which are zero on the boundary of Ω and continuously differentiable on the closure of Ω, is a subset of H. We define the H to be the completion of $C_0^1(\bar{\Omega})$ in L^2 under this norm.

Definition A.0.8 *In an inner product space, a collection of vectors $\{\psi_\alpha\}$ is orthonormal if $||\psi_\alpha|| = 1$ for all α and $\langle \psi_\alpha, \psi_\beta\rangle = 0$ for all $\alpha \neq \beta$.*

Definition A.0.9 *On a region $\Omega \subset \mathbf{R}$, Δ is the linear operator such that $\Delta u = u_{xx} + u_{yy}$, called the Laplacian Operator.*

Definition A.0.10 *If there is a unique z such that $u'(x)(y) = \langle z, y\rangle_2$ then $\nabla_2 u = z$. The funciton $\nabla_2 u$ is said to be the L^2 gradient of u. If there is a unique z such that $u'(x)(y) = \langle z, y\rangle_H$ then $\nabla_H u = z$. The function $\nabla_H u$ is said to be the Sobolev gradient of u.*

Definition A.0.11 *Given a function $J : H \to \mathbf{R}$, we define critical points of J as functions u such that $J'(u)(v) = 0$ for all $v \in H$.*

Definition A.0.12 *The Morse index (MI) of a critical point is the number of negative eigenvalues of the Hessian matrix. Roughly speaking, it is the number of linearly independent "down" directions in function space of a functional. For example, $x^2 + y^2 - z^2$ and $x^2 - y^2 - z^2$ have Morse index 1 and Morse index 2 critical points at $(0, 0, 0)$, respectively.*

Definition A.0.13 *Steepest descent and ascent techniques are used to find local minimums and maximums. Let $J : X \to \mathbf{R}$. Recall that the gradient $\nabla J(u)$ points in the direction of steepest ascent. To perform steepest ascent, using k as a counter and $\delta > 0$ as a stepsize, we repeat the following step until $||\nabla J(u)|| < \epsilon$:*

$$u^{k+1} = u^k + \delta \nabla J(u^k).$$

In order to perform steepest descent, we simply make $\delta < 0$. In theory, for many functions J one can find an optimal stepsize δ which is used to achieve convergence as quick as possible.

Appendix B

ODE Reduction Code

```
//-------------------------------------------------------------
//Reduction Code for ODE
//Objective: Solve y'' + f(y) = 0 on [0,1] where y(0)=y(1)=0
//          Do this by finding critical points of J(u).
//          One critical point u satisfies:
//          J(u) = max min J(x+y)
//                 x   y
//         where x is in X1 and y is in X2.
//-------------------------------------------------------------

#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <iomanip.h>

const double pi = 3.141592654;
const double pi2 = pi*pi;
const double pi2_div2 = .5*pi2;
const int n = 100;          //number of divisions
const int m = 100;          //number of eigenfunctions in basis
const double delta = .1;    //step size
const double tol = 0.000001;
const double kval = 1;    //value to be assigned to initial
                          //coefficients
const int k = 2;          //crossing eval number
```

```
const int k_next = 3;
const int max_its = 10000;

//t = 0.5 means that f'(infinity) is halfway between lambda_k
//and lambda_k+1
const double t = 0.5;
const double fpi = (double)(1-t)*pi2*k*k
                 + (double)t*pi2*k_next*k_next;   //f'(inf)
const double fp0 = 0;   //f'(0)

void fourier(double a[m+1], double u[n+1]);
double lam(int i);         //(pi*k)^2
double psi(int i, double x);//sin(1.414*k*pi*x)
double f(double x);         //f(x)
double F(double x);         //F(x) (f'(x) = F(x))
double J(double a[m+1], double u[n+1]);       //J(u)
double Jp(double a[m+1], double u[n+1], int j);
//J'(u)(psi(j,x))
void grad2(double a[m+1], double u[n+1], double g[m+1]);
//performs the L2 gradient
double norm(double u[m+1]);
void coefficients(double a[m+1], double u[n+1]);
void error(double a[m+1], double u[n+1], ofstream & fout);

void main()
{
   double cnvg = 1;   //Used to test for convergence
   double a[m+1];      //Fourier coefficients
   double u[n+1];
   double g[m+1];      //L2 Gradient of u
   ofstream fout;
   fout.open("ode.txt");   //Output file

   //Set up initial fourier coefficients
   for (int i = 0; i <= n; i++) u[i] = 0;
      for (i = 0; i <= m; i++)
      {
         if (i == k) a[i] = kval;
```

```
         else a[i] = 0;
      }

   fourier(a,u);   //Fourier expansion

   //Main loop
   for (int iteration = 1; iteration <= max_its; iteration++)
   {
      grad2(a,u,g);   //Set g = L2 Gradient of u

      //Here we convert to the Sobolev gradient, as well as
      //setting g to be negative in the X components
      for (i = 1; i <= k; i++)
         g[i] = -g[i]/lam(i);
      for (i = k+1; i <= m; i++)
         g[i] = g[i]/lam(i);
      //This step will perform steepest ascent in X,
      //steepest descent in Y
      for (i = 1; i <= m; i++)
         a[i]-= delta*g[i];

      fourier(a,u);   //Fourier expansion

      cnvg = norm(g); //Check size of norm - if norm is small
                      //we must be near a critical point.

      //If ||g|| is small enough, stop and calculate residual
      if (cnvg < tol)
      {
         iteration = max_its + 1;   //exit loop
         error (a, u, fout);
      }
   }
   fout.close();
}

//Fourier expansion - the function sets
//u = sum(a_i psi_i)
```

```
void fourier(double a[m+1], double u[n+1])
{
   double x;
   for (int j = 0; j <=n; j++)
   {
      u[j] = 0;
      x = j/(double)n;
      for (int i = 1; i <=m; i++)
         u[j] += a[i]*psi(i,x);
   }
}

//(Pi*k)^2
double lam(int i)
{
   return (pi*pi*i*i);
}

//Sin(1.414*k*pi*x)
double psi(int i, double x)
{
   return (1.4142136*sin(i*pi*x));
}

//Function f: recall r = f'(inf), s = f'(0)
double f(double x)
{
   if (x >= 0)
      return (fpi*x + fp0*log(1+x) - fpi*log(1+x));
   else
      return (fpi*x - fp0*log(1-x) + fpi*log(1-x));
}

//F(x) = int_0^x f(s)ds
double F(double x)
{
   if (x >= 0)
      return(.5*fpi*x*x + (fpi-fp0)*x
```

```
      +  log(1+x)*(fp0+fp0*x-fpi-fpi*x));
   else
      return(.5*fpi*x*x + (-fpi+fp0)*x
      + log(1-x)*(-fpi+fp0+fpi*x-fp0*x));
}

//Want to return: (1/2)sum(a_i^2 lam_i) - int_0^1 F(u)dx
double J(double a[m+1], double u[n+1])
{
   double sum = 0;
   double sum2 =0;
   for (int i = 1; i <= m; i++)
      sum+=.5*a[i]*a[i]*lam(i);

//integrate from 0 to 1: F(x)
   for (i = 0; i < n; i++)
      sum2 += F(u[i]);
   sum2 = sum2/(double)n;
   return sum - sum2;
}

//Want to return: a_j lam_j - \int_0^1 psi_j f(u)dx
double Jp(double a[m+1], double u[n+1], int j)
{
   double sum = a[j]*lam(j);
   double x = 0;
   int i;

   double sum2 = 0;
   if (integration == 0)//left hand sum
   {
      for (i = 0; i < n; i++)
      {
         x = i/(double)n;
         sum2 += psi(j,x)*f(u[i]);
      }
      sum2 = sum2/(double)n;
   }
```

```
   return sum - sum2;
}

//For each i, set gradient_i = sum(J'(u)(psi_i))
void grad2(double a[m+1], double u[n+1], double g[m+1])
{
   for (int i = 1; i <= m; i++)
      g[i] = Jp(a, u, i);
}

double norm(double u[m+1])
{
   double sum = 0;
   for (int i = 1; i <= m; i++)
      sum += pow(u[i],2);
   return sqrt(sum);
}

//Sets coefficients a, given a function u. (used for some
//tests)
void coefficients(double a[m+1], double u[n+1])
{
   double x;
   for (int r = 1; r <= m; r++)
   {
      a[r] = 0;
         //for each a[r], integrate u(x)*psi(r,x)
      for (int i = 0; i < n; i++)
      {
         x = i/(double)n;
         a[r] += u[i]*psi(r,x);
      }
      a[r] = a[r]/(double)n;
   }
}

//Output estimated residuals to file
void error(double a[m+1], double u[n+1], ofstream & fout)
```

```
{
   //Output u
   fout << "u====================" << endl;
   for (int i = 0; i <= n; i++)
      fout << i/(double)n << "   " << setprecision(10)
      << u[i] << endl;

   //Output coefficients
   fout << "a====================" << endl;
   for (i = 1; i <= m; i++)
      fout << a[i] << endl;

   double error;
   double error_2 = 0;
   double error_max = 0;
   double u_2 = 0;
   double u_max = 0;
   //Calculate residuals by performing divided differences
   //(L2 norm and max norms)
   for (int i = 1; i < n; i++)
   {
      error = (u[i+1] - 2*u[i] + u[i-1])*n*n;
      error = fabs(error + f(u[i]));
      if (error > error_max) error_max = error;
      error_2 += error*error;

      //Calculate ||u|| (L2 norm and max norms)
      u_2 += u[i]*u[i];
      if (fabs(u[i]) > u_max) u_max = fabs(u[i]);
   }
   error_2 = error_2/(double)n;
   u_2 += u[n]*u[n];
   u_2 = u_2/(double)n;

   cout << "n = " << n << ", m = " << m << ", tol = " << tol
        << endl;
   cout << "f'(0) = " << fp0 << endl;
   cout << "f'(inf) = " << fpi << endl;
```

```
    cout << "||u||2 = " << sqrt(u_2) << endl;
    cout << "||u||inf = " << u_max << endl;
    cout << "||err||2 = " << sqrt(error_2) << ";  "
         << sqrt(f_2) << endl;
    cout << "||err||inf = " << error_max << ";  " << f_max
         << endl;
}
```

Appendix C

PDE Reduction Code

```
//-------------------------------------------------------------
//Objective: Solve laplacian(u) + f(u) = 0 on [0,1]x[0,1]
//          Do this by finding critical points of J(u).
//          One critical point u satisfies:
//          J(u) = max min J(x+y)
//                  x   y
//         where x is in X1 and y is in X2.
//-------------------------------------------------------------

#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <iomanip.h>
#include <time.h>

const double pi = 3.141592654;
const double pi2 = pi*pi;
const double pi2_div2 = .5*pi2;
const int n = 20;         //number of divisions
const int m = 20;         //number of eigenfunctions in basis
const int n2 = n*n;
const double delta1 = .7;   //steepest ascent step size in X
const double delta2 = .7;   //steepest descent step size in Y
const double tol = 0.000001;
const double kval = 1;  //value at k
```

```
const int k = 2;   //k can be 2,5,8,10,13,17,18,20,25,26,29,
                   //32,34,37
                   //(its a sum of 2 perfect squares)
const int k_next = 5;
const int max_its = 100000;
const double fp0 = 0;   //f'(0)

//t = 0.5 means that f'(infinity) is halfway between lambda_k
//and lambda_k+1
const double t= .5;
const double fpi = (double)(1-t)*pi2*k + (double)t*pi2*k_next;

void fourier(double a[m+1][m+1], double u[n+1][n+1]);
double lam(int i, int j);
double psi(int i, int j, double x, double y);
double f(double x);
double fp(double x);
double F(double x);
double J(double a[m+1][m+1], double u[n+1][n+1]);
double Jp(double a[m+1][m+1], double u[n+1][n+1], int j);
void grad2(double a[m+1][m+1], double u[n+1][n+1],
           double g[m+1][m+1]);
double norm(double g[m+1][m+1]);
void print_error (double a[m+1][m+1], double u[n+1][n+1],
                  ofstream & fout);

void main()
{
   double cnvg = 1;
   double a[m+1][m+1];   //Fourier coefficients
   double u[n+1][n+1];
   double g[m+1][m+1];   //L2 gradient of u
   double u_max;
   ofstream fout;
   fout.open("Reduction.txt");   //output file

   for (int i = 0; i <=n; i++)
      for (int j = 0; j <=n; j++)
```

```
      u[i][j] = 0;

//Set up initial Fourier coefficients
if (k==2)
{
   for (int r = 0; r <= m; r++)
   for (int s = 0; s <= m; s++)
   {
      if ((r*r + s*s) == k) a[r][s] = kval;
      else a[r][s] = 0;
   }
}
else if (k==5)
{
   for (int r = 0; r <= m; r++)
      for (int s = 0; s <= m; s++)
      {
         if ((r == 2) && (s == 1)) a[r][s] = kval;
         else a[r][s] = 0;
      }
}

fourier(a,u);   //Fourier expansion

//Main loop
for (int iteration = 1; iteration < max_its; iteration++)
{
   grad2(a,u,g);   //set g = L2 gradient of u

   //Here we convert to the Sobolev gradient, as well as
   //setting g to be negative in the X components
   for (int r = 1; r <= m; r++)
   {
      for (int s = 1; s <= m; s++)
      {
         if (r*r + s*s <= k)
            g[r][s] = -g[r][s]/lam(r,s);
         else
```

```
                g[r][s] = g[r][s]/lam(r,s);
            }
         }

         //This step will perform steepest ascent in X,
         //steepest descent in Y
         for (r = 1; r <= m; r++)
            for (int s = 1; s <= m; s++)
               a[r][s] -= delta1*g[r][s];

         fourier(a,u);   //Fourier expansion

         cnvg = norm(g); //Check size of norm - if norm is small
                         //we must be near a critical point.

         //If ||g|| is small enough, stop and calculate residual
         if (cnvg < tol)
         {
            iteration = max_its + 1;
            print_error(a,u,fout);
         }
      }
      fout.close();
}

//Fourier expansion - the function sets
//u = sum(a_i psi_i)
void fourier(double a[m+1][m+1], double u[n+1][n+1])
{
   double x,y;
   for (int i = 0; i <= n; i++)
   {
      x = i/(double)n;
      for (int j = 0; j <= n; j++)
      {
         u[i][j] = 0;
         y = j/(double)n;
         for (int r = 1; r <= m; r++)
```

```
            for (int s = 1; s <= m; s++)
               u[i][j] += a[r][s]*psi(r,s,x,y);
      }
   }
}

//pi^2(i^2+j^2)
double lam(int i, int j)
{
   return (pi*pi*(i*i+j*j));
}

//2 Sin(i Pi x)sin(i Pi y)
double psi(int i, int j, double x, double y)
{
   return (2*sin(i*pi*x)*sin(j*pi*y));
}

//Function f: fpi = f'(inf), fp0 = f'(0)
double f(double x)
{
   if (x >= 0)
      return (fpi*x + fp0*log(1+x) - fpi*log(1+x));
   else
      return (fpi*x - fp0*log(1-x) + fpi*log(1-x));
}

//F(x) = int_0^x f(s)ds
double F(double x)
{
   if (x >= 0)
      return(.5*fpi*x*x + (fpi-fp0)*x
             + log(1+x)*(fp0+fp0*x-fpi-fpi*x));
   else
      return(.5*fpi*x*x + (-fpi+fp0)*x
             + log(1-x)*(-fpi+fp0+fpi*x-fp0*x));
}
```

```
//Want to return: (1/2)sum(a_i^2 lam_i) - int_0^1 F(u)dx
double J(double a[m+1][m+1], double u[n+1][n+1])
{
   double sum = 0;
   double integral = 0;
   for (int r = 1; r <= m; r++)
      for (int s = 1; s <=  m; s++)
         sum+=.5*a[r][s]*a[r][s]*lam(r,s);

//integrate from 0 to 1: F(x)
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         integral -= F(u[i][j]);
   integral = integral/(double)n2;
   return sum + integral;
}

//Want to return: a_j lam_j - \int_0^1 psi_j f(u)dx
double Jp(double a[m+1][m+1], double u[n+1][n+1], int r,
          int s)
{
   double x = 0;
   double y = 0;
   double sum = 0;
   for (int i = 0; i < n; i++)
   {
      x = i/(double)n;
      for (int j = 0; j < n; j++)
      {
         y = j/(double)n;
         sum += psi(r,s,x,y)*f(u[i][j]);
      }
   }
   return a[r][s]*lam(r,s) - sum/(double)n2;
}

//For each i, set gradient_i = sum(J'(u)(psi_i))
void grad2(double a[m+1][m+1], double u[n+1][n+1],
```

```
           double g[m+1][m+1])
{
   for (int r = 1; r <= m; r++)
      for (int s = 1; s <= m; s++)
         g[r][s] = Jp(a, u, r, s);
}

double norm(double g[m+1][m+1])
{
   double sum = 0;
   for (int r = 1; r <= m; r++)
      for (int s = 1; s <= m; s++)
         sum += pow(g[r][s],2);
   return sqrt(sum);
}

//Output estimated residuals to file
void print_error(double a[m+1][m+1], double u[n+1][n+1],
                 ofstream & fout)
{
   double x, y, error;
   double err_2 = 0;   //L2 norm of residual
   double err_max = 0;   //inf norm of residual
   double u_2 = 0;      //L2 norm of u
   double u_max = 0;   //inf norm of u
   for (int i = 0; i < n; i++)
   {
      x = i/(double)n;
      for (int j = 0; j < n; j++)
      {
         y = j/(double)n;

         error = 0;
         //Set error = -Sum(a_i lam_i psi_i)
         for (int r = 1; r <= m; r++)
         {
            for (int s = 1; s <= m; s++)
            {
```

```
                error -= a[r][s]*lam(r,s)*psi(r,s,x,y);
             }
          }
          error = error + f(u[i][j]);
          if (err_max < fabs(error)) err_max = fabs(error);
          err_2 += error*error;

          u_2 += u[i][j]*u[i][j];
          if (u_max < fabs(u[i][j])) u_max = fabs(u[i][j]);
       }
    }

    cout << "n = " << n << ", m = " << m << ", tol = "
         << tol << endl;
    cout << "J(u) = " << J(a,u) << endl;
    cout << "f'(0) = " << fp0 << endl;
    cout << "f'(inf) = " << fpi << endl;
    cout << "||u||2 = " << sqrt(u_2/(double)n2) << endl;
    cout << "||u||inf = " << u_max << endl;
    cout << "||err||2 = " << sqrt(err_2/(double)n2) << endl;
    cout << "||err||inf = " << err_max << endl;
}
```

Appendix D

ODE One-Sign Algorithm Code

```
//-------------------------------------------------------------
//One-sign Algorithm Code for ODE
//Objective: Solve y'' + f(y) = 0 on [0,1] where y(0)=y(1)=0
//   in each iteration $u$ is projected onto
//   S = {u in H - {0}: J'(u)(u) = 0 }
//   after which one takes a step in the $-\nabla J(u)$
//   direction.
//-------------------------------------------------------------

#include <iostream.h>
#include <fstream.h>
#include <math.h>

const int m = 100;   //number of eigenfunctions in basis
const int n = 100;   //number of divisions
const int max_its = 5000;
const double tol1 = 0.000001;   //tol for projection step
const double tol2 = 0.000001;   //tol for steepest descent
const int k = 1;   //crossing eval number
const int k_next = 2;
const double kval = 1;
const double delta1 = 0.1;   //step size for projection step
const double delta2 = 0.1;   //step size for steepest descent
const double pi = 3.14159;
const double pi2 = pi*pi;
```

```
const double s2 = 1.414;

//t = 0.5 means that f'(infinity) is halfway between
//lambda_k and lambda_k+1
const double t = .5;
const double fpi = (double)(1-t)*pi2*k*k
                + (double)t*pi2*k_next*k_next;   //f'(inf)
const double fp0 = 0;   //f'(0)

void ps(double a[m+1], double u[n+1], ofstream & fout);
double pgrad_J(double a[m+1], double u[n+1]);
double Jp(double a[m+1], double u[n+1], int r);
void grad2(double a[m+1], double u[n+1], double g[m+1]);
void fourier(double a[m+1], double u[n+1]);
double norm(double a[m+1]);
double lam(int i);
double psi(int i, double x);
double f(double x);
double F(double x);
void print_u(double u[n+1], ofstream & fout);
void print_error(double a[m+1], double u[n+1],
                ofstream & fout);

void main()
{
   double u[n+1];
   double a[m+1];   //Fourier coefficients
   double g[m+1];   //L2 Gradient of u
   double cnvg;     //Used to test for convergence

   ofstream fout;
   fout.open("ode-os.txt");   //output file

   //Initialize u
   for (int i = 0; i <=n; i++)
      u[i] = 0;

   //Set initial Fourier coefficients
```

```
   for (int r = 0; r <= m; r++)
   {
      if (r == k) a[r] = kval;
      else a[r] = 0;
   }
   fourier(a,u);   //Fourier expansion

   //Main loop
   for (int iteration = 1; iteration < max_its; iteration++)
   {

      ps(a,u,fout);   //project u onto S
      fourier(a,u);   //Fourier expansion
      grad2(a,u,g);   //Set g = L2 Gradient of u

      //Steepest descent step
      for (r = 1; r <= m; r++)
      {
         g[r] = g[r]/(lam(r));   //convert to Sobolev gradient
         a[r] = a[r] - delta1*g[r];
      }
      fourier(a,u);   //Fourier expansion

      cnvg = norm(g); //Check size of norm - if norm is
                  //small we must be near a critical point.

      //If ||g|| is small enough, stop and calculate residual
      if ((iteration > 1)&&(cnvg < tol1))
      {
         iteration = max_its + 1;   //exit loop
         print_error(a, u, fout);
      }
   }
}

//Fourier expansion - the function sets
//u = sum(a_i psi_i)
void fourier(double a[m+1], double u[n+1])
```

```
{
   double x;
   for (int i = 0; i <= n; i++)
   {
      u[i] = 0;
      x = i/(double)n;
      for (int r = 1; r <= m; r++)
         u[i] += a[r]*psi(r,x);
   }
}

//Projects u onto S = {u in H - {0}: J'(u)(u) = 0 }
void ps(double a[m+1], double u[n+1], ofstream & fout)
{
   double t, cnvg;
   double old_a[m+1];
   for (int it = 1; it < max_its; it++)
   {
      cnvg = 0;   //keeps track of ||a^{m+1} - a^m||
      t = pgrad_J(a,u);   //Projection of gradient onto u = tu

      //To perform steepest ascent in ray direction
      for (int r = 1; r <= m; r++)
      {
         old_a[r] = a[r];
         a[r] = a[r]*(1 + delta2*t);
         cnvg += pow(old_a[r] - a[r], 2);
      }
      cnvg = sqrt(cnvg);

      //If ||a^{m+1} - a^m|| is small, exit loop
      if ((it > 1)&&(cnvg < tol2)) it = max_its + 1;

      fourier(a,u);   //Fourier expansion
   }
}

//Calculate (Sum(a_i^2 lam_i) - integral u f(u))
```

```
//          /Sum(a_1^2 lam_i)
double pgrad_J(double a[m+1], double u[n+1])
{
   double sum = 0;
   double integral = 0;
   for (int r = 1; r <= m; r++)
      sum += a[r]*a[r]*lam(r);

//integrate from 0 to 1: xf(x)
   for (int i = 0; i < n; i++)
      integral += u[i]*f(u[i]);
   integral = integral/(double)n;
   return (sum - integral)/sum;
}

//Want to return: a_j lam_j - \int_0^1 psi_j f(u)dx
double Jp(double a[m+1], double u[n+1], int r)
{
   double integral = 0;
   double x;

   for (int i = 0; i < n; i++)
   {
      x = i/(double)n;
      integral += psi(r,x)*f(u[i]);
   }
   integral = integral/(double)n;
   return a[r]*lam(r) - integral;
}

//For each i, set gradient_i = sum(J'(u)(psi_i))
void grad2(double a[m+1], double u[n+1], double g[m+1])
{
   for (int r = 1; r <= m; r++)
      g[r] = Jp(a, u, r);
}

//(Pi*k)^2
```

```
double lam(int i)
{
   return (pi*pi*i*i);
}

//Sin(1.414*k*pi*x)
double psi(int i, double x)
{
   return (s2*sin(i*pi*x));
}

//Function f: recall fpi = f'(inf), fp0 = f'(0)
double f(double x)
{
   if (x >= 0)
      return (fpi*x + fp0*log(1+x) - fpi*log(1+x));
   else
      return (fpi*x - fp0*log(1-x) + fpi*log(1-x));
}

//F(x) = int_0^x f(s)ds
double F(double x)
{
   if (x >= 0)
      return(.5*fpi*x*x + (fpi-fp0)*x
             + log(1+x)*(fp0+fp0*x-fpi-fpi*x));
   else
      return(.5*fpi*x*x + (-fpi+fp0)*x
             + log(1-x)*(-fpi+fp0+fpi*x-fp0*x));
}

double norm(double a[m+1])
{
   double sum = 0;
   for (int r = 1; r <= m; r++)
      sum += pow(a[r],2);
   return sqrt(sum);
}
```

```
//Output estimated residuals to file
void print_error(double uu[m+1], double u[n+1],
                 ofstream & fout)
{
   double error, x;
   double error_2 = 0;
   double error_max = 0;
   double u_2 = 0;
   double u_max = 0;
   for (i = 1; i < n; i++)
   {
      //Find error using divided differences,
      //both in L2 and inf norms
      error = (u[i+1] - 2*u[i] + u[i-1])*n*n;
      error = fabs(error + f(u[i]));
      if (error > error_max) error_max = error;
      error_2 += error*error;

      //Calculate ||u||, both in L2 and inf norms
      u_2 += u[i]*u[i];
      if (fabs(u[i]) > u_max) u_max = fabs(u[i]);
   }
   error_2 = error_2/(double)n;
   f_2 = f_2/(double)n;
   u_2 += u[n]*u[n];
   u_2 = u_2/(double)n;

   cout << "m = " << m << ", n = " << n << ", tol1 = "
        << tol1 << endl;
   cout << "f'(0) = " << fp0 << endl;
   cout << "f'(inf) = " << fpi << endl;
   cout << "||u||2 = " << sqrt(u_2) << endl;
   cout << "||u||inf = " << u_max << endl;
   cout << "||err||2 = " << sqrt(error_2) << ";  "
        << sqrt(f_2) << endl;
   cout << "||err||inf = " << error_max << ";  " << f_max
        << endl;}
```

Appendix E

Shooting Method Code, $k = 1$

The following code was implemented using *Mathematica*. In order to find the $k = 2$ solution, one only needs to change values for $f'(\infty)$ as well as test that $y(1) > 0$ instead of $y(1) < 0$.

```
fpi = 2.5*Pi^2;
fp0 = 0;
f[x_] := If[x >= 0, Return[fpi*x + fp0*Log[1 + x]
                           - fpi*Log[1 + x]],
      fpi*x - fp0*Log[1 - x] + fpi*Log[1 - x]];

n = 1001; (*number of gridpoints*)
h = 1/1000; (*size of division*)
y1 = Table[0, {i, 1, n}]; (*y(x)*)
u = Table[0, {i, 1, n}]; (*y'(x)*)
For[d = 5, d < 34, d += .1,
   u[[1]] = d; (*set d = y'(0)*)
   y1[[1]] = 0; (*set y(0) = 0*)
   For[i = 1, i < n, i++,
      y1[[i + 1]] = y1[[i]] + h*u[[i]];
      u[[i + 1]] = u[[i]] - h*f[y1[[i + 1]]];
      ];
   (*test if y(1) < 0*)
   If [y1[[n]] < 0,
      Print[InputForm[d], " ", y1[[n]]];
      d = 100; (*exit loop*)      ];  ];
```