MCLITE: AN ADAPTIVE MULTILEVEL FINITE ELEMENT MATLAB PACKAGE FOR SCALAR NONLINEAR ELLIPTIC EQUATIONS IN THE PLANE

M. HOLST

Abstract. This paper is a user's manual for MCLite, an adaptive multilevel finite element MATLAB package for solving scalar nonlinear elliptic equations in the plane. MCLite is a two-dimensional MATLAB prototype of a more complete two- and three-dimensional ANSI-C code called MC, which expends MCLite's capabilities to elliptic systems of tensor equations on 2- and 3-manifolds. Both codes share the same core geometry datastructures and overall design; they even share the same input file format. This allows a user to work with both codes, perhaps prototyping ideas in MCLite using MATLAB, and then reimplementing the idea more carefully in MC using ANSI-C.

1. Introduction

This paper is a user's manual for MCLite, an adaptive multilevel finite element MATLAB package for solving scalar nonlinear elliptic equations in the plane. MCLite is a two-dimensional MATLAB prototype of a more complete two- and three-dimensional ANSI-C code called MC, which expends MCLite's capabilities to elliptic systems of tensor equations on 2- and 3-manifolds. Both codes share the same core geometry datastructures and overall design; they even share the same input file format. This allows a user to work with both codes, perhaps prototyping ideas in MCLite using MATLAB, and then reimplementing the idea more carefully in MC using ANSI-C.

The outline of the manual is as follows. In §2, we look at general formulations of scalar elliptic equations, deriving the nonlinear weak form and its linearization bilinear form that play key roles in MCLite and MC. As an example, in §3 we derive the weak form and the linearization form for a nonlinear Poisson-like scalar equation. In §4, we outline the structure of adaptive multilevel finite element methods in general, and discuss the overall structure of the MCLite and MC implementations. In §5, we describe the MCLite implementation in more detail, including the core geometry datastructures and design features that are common to both MC and MCLite. We point to the web site where MCLite may be found in §6. In the Appendix, we list the primary MCLite routines, and give an example illustrating the use of MCLite to solve a problem. We also and describe how the geometry datastructures, and the algorithms for traversing and modifying the datastructures, are implemented in MATLAB.

2. Nonlinear elliptic equations, weak forms, and linearizations

In this section we will first give an overview of weak formulations of nonlinear elliptic equations, as required for use of finite element methods. Some standard references for this material are [25, 27].

Let $\Omega \subset \mathbb{R}^d$ be an open set, and let $\partial\Omega$ denote the boundary, which can be thought of as a set in $\mathbb{R}^{d-1}$ (more accurately, it is a d-manifold). We will consider the case where $\partial\Omega$ is formed as the union of two disjoint sets $\partial_D\Omega$ and $\partial_N\Omega$, which can be summarized mathematically as:

$$\partial_D\Omega \cup \partial_N\Omega = \partial\Omega, \qquad \partial_D\Omega \cap \partial_N\Omega = \emptyset.$$

For technical reasons, it will be important that one of the boundary sets be of non-trivial size, written mathematically as $\text{meas}_{d-1}(\partial_D\Omega) > 0$. This condition is related to the type of boundary condition that will be imposed on the set $\partial_D\Omega$, and it will always hold for the boundary conditions arising in typical boundary value problems. Consider now the following nonlinear scalar elliptic equation on Ω:

$$-\nabla \cdot a(x, \hat{u}(x), \nabla\hat{u}(x)) + b(x, \hat{u}(x), \nabla\hat{u}(x)) = 0 \text{ in } \Omega, \tag{2.1}$$

$$n(x) \cdot a(x, \hat{u}(x), \nabla\hat{u}(x)) + c(x, \hat{u}(x), \nabla\hat{u}(x)) = 0 \text{ on } \partial_N\Omega, \tag{2.2}$$

$$\hat{u}(x) = g_D(x) \text{ on } \partial_D\Omega, \tag{2.3}$$

where $n(x) : \partial_N \Omega \mapsto \mathbb{R}^d$ is the unit normal to $\partial\Omega$, and where

$$a : \Omega \times \mathbb{R} \times \mathbb{R}^d \mapsto \mathbb{R}^d, \quad b, c : \Omega \times \mathbb{R} \times \mathbb{R}^d \mapsto \mathbb{R}, \quad g_D : \partial_D \Omega \mapsto \mathbb{R}.$$

The above form of the equation is sometimes referred to as the *strong form*, in that the solution is required to be twice differentiable for the equation to hold in the classical sense. To produce a weak formulation, which is more suitable for finite element methods in that it will require that fewer derivatives of the solution exist, we first choose or construct a *test* space of functions. For second order scalar elliptic equations of this form, the appropriate test space is $H^1(\Omega)$, the *Sobolev* space of functions which are differentiable one time under the integral. This function space is simply the set of all scalar-valued functions over the domain Ω for which the following integral (the energy norm, or H^1-norm) is always finite:

$$\|u\|_{H^1(\Omega)} = \left(\int_\Omega |\nabla u|^2 + |u|^2 dx \right)^{1/2}.$$

In other words, the space of functions $H^1(\Omega)$ is defined as:

$$H^1_{0,D}(\Omega) = \left\{ \; u : \|u\|_{H^1(\Omega)} < \infty \; \right\}.$$

(To be precise, all integrals here and below must be interpreted in the Lebesgue rather than Riemann sense, but we will ignore the distinction here.) It is important that the test space satisfy a zero boundary condition on the part of the boundary $\partial_D\Omega$ on which the Dirichlet boundary condition (2.3) holds, so that in fact we choose as the test space the following subspace of $H^1(\Omega)$:

$$H^1_{0,D}(\Omega) = \left\{ \; u \in H^1(\Omega) : u = 0 \text{ on } \partial_D \Omega \; \right\}.$$

Now, multiplying our elliptic equation by a test function $v \in H^1_{0,D}(\Omega)$ produces:

$$\int_\Omega \left(-\nabla \cdot a(x, \hat{u}, \nabla \hat{u}) + b(x, \hat{u}, \nabla \hat{u}) \right) v \; dx = 0,$$

which becomes, after applying Green's integral identities,

$$\int_\Omega a(x, \hat{u}, \nabla \hat{u}) \cdot \nabla v \; dx - \int_{\partial\Omega} v a(x, \hat{u}, \nabla \hat{u}) \cdot n \; ds + \int_\Omega b(x, \hat{u}, \nabla u) v \; dx = 0. \tag{2.4}$$

The boundary integral above is reformulated using the Robin boundary condition (2.2) as follows:

$$\int_{\partial\Omega} v a(x, \hat{u}, \nabla \hat{u}) \cdot n \; ds = \int_{\partial_D\Omega} v a(x, \hat{u}, \nabla \hat{u}) \cdot n \; ds + \int_{\partial_N\Omega} v a(x, \hat{u}, \nabla \hat{u}) \cdot n \; ds \tag{2.5}$$

$$= 0 - \int_{\partial_N\Omega} v c(x, \hat{u}, \nabla \hat{u}) \; ds. \tag{2.6}$$

If the boundary function g_D is smooth enough, there is a mathematical result called the Trace Theorem [1] which guarantees the existence of a function $u_D \in H^1(\Omega)$ such that $g_D = u_D \,|_{\partial_D\Omega}$. (We will not be able to construct such a *trace* function u_D in practice, but we will be able to approximate it as accurately as necessary in order to use this weak formulation in finite element methods.) Employing such a function $u_D \in H^1(\Omega)$, we define the following affine or translated Sobolev space:

$$H^1_{g,D}(\Omega) = \left\{ \; \hat{u} \in H^1(\Omega) \;\; | \;\; u + u_D, \; u \in H^1_{0,D}(\Omega), \; g_D = u_D \,|_{\partial_D\Omega} \; \right\}.$$

It is easily verified that the solution $\hat{u}$ to the problem (2.1)–(2.3), if one exists, lies in $H^1_{g,D}(\Omega)$, although unfortunately $H^1_{g,D}(\Omega)$ is not a vector space, since it is not linear. (Consider that if $u, v \in H^1_{g,D}(\Omega)$, it holds that $u + v \notin H^1_{g,D}(\Omega)$.) It is important that the problem be phrased in terms of vector spaces such as $H^1_{0,D}(\Omega)$, in order that certain analysis tools and concepts be applicable. Therefore, we will do some additional transformation of the problem.

So far, we have shown that the solution to the original problem (2.1)–(2.3) also solves the following problem:

$$\text{Find } \hat{u} \in H^1_{g,D}(\Omega) \text{ such that } \; < \hat{F}(\hat{u}), v >= 0 \quad \forall v \in H^1_{0,D}(\Omega), \tag{2.7}$$

where from equations (2.4) and (2.5), the scalar-valued function of $\hat{u}$ and v (also called a *form*), nonlinear in $\hat{u}$ but linear in v, is defined as:

$$< \hat{F}(\hat{u}), v >= \int_\Omega a(x, \hat{u}, \nabla\hat{u}) \cdot \nabla v + b(x, \hat{u}, \nabla\hat{u}) v \; dx + \int_{\partial_N \Omega} c(x, \hat{u}, \nabla\hat{u}) v \; ds.$$

Since we can write the solution $\hat{u}$ to equation (2.7) as $\hat{u} = u + u_D$ for a fixed u_D satisfying $g_D = u_D\,|_{\partial_D\Omega}$, we can rewrite the equations completely in terms of u and a new nonlinear form $< F(\cdot), \cdot >$ as follows:

$$\text{Find } u \in H^1_{0,D}(\Omega) \text{ such that } \; < F(u), v >= 0 \quad \forall v \in H^1_{0,D}(\Omega), \tag{2.8}$$

$$\begin{aligned} < F(u), v >= & \int_\Omega a(x, [u + u_D], \nabla[u + u_D]) \cdot \nabla v + b(x, [u + u_D], \nabla[u + u_D]) v \; dx \\ & + \int_{\partial_N \Omega} c(x, [u + u_D], \nabla[u + u_D]) v \; ds. \end{aligned} \tag{2.9}$$

Clearly, the "weak" formulation of the problem given by equation (2.8) imposes only one order of differentiability on the solution u, and only in the weak sense (under an integral). Under suitable growth restrictions on the nonlinearities b and c, it can be shown that this weak formulation makes sense, in that the form $< F(\cdot), \cdot >$ is finite for all arguments. Moreover, it can be shown under somewhat stronger assumptions, including the condition that $\text{meas}_{d-1}(\partial_D\Omega) > 0$, that the weak formulation is well-posed, in that there exists a unique solution depending continuously on the problem data.

To apply a Newton iteration to solve this nonlinear problem numerically, we will need a linearization of some sort. Rather than discretize and then linearize the discretized equations, we will exploit the fact that with projection-type discretizations such as the finite element method, these operations actually commute; we can first linearize the differential equation, and then discretize the linearization in order to employ a Newton iteration. To linearize the weak form operator $< F(u), v >$, a bilinear linearization form $< DF(u)w, v >$ is produced as its directional (variational, Gateaux) derivative as follows:

$$< DF(u)w, v >= \frac{d}{dt} < F(u + tw), v > \Big|_{t=0}$$

$$= \frac{d}{dt} \int_\Omega (a\nabla[u + u_D + tw] \cdot \nabla v + b(x, [u + u_D + tw])v - fv) \; dx + \int_{\partial_N \Omega} (c[u + u_D + tw] - g_N)\, v \; ds|_{t=0}$$

$$= \int_\Omega \left(a\nabla w \cdot \nabla v + \frac{\partial b(x, u + u_D)}{\partial u} wv \right) \; dx + \int_{\partial_N \Omega} cwv \; ds.$$

This scalar-valued function of the three arguments u,v, and w, is linear in w and v, but possibly nonlinear in u. For fixed (or frozen) u, it is referred to as a bilinear form (linear in each of the remaining arguments w and v). We will see shortly that the nonlinear weak form $< F(u), v >$, and the associated bilinear linearization form $< DF(u)w, v >$, together with a continuous piecewise polynomial subspace of the solution space $H^1_{0,D}(\Omega)$, are all that are required to employ the finite element method for numerical solution of the original elliptic equation.

3. A nonlinear Poisson equation example: weak form and linearization

Here is an example derivation of the weak formulation of a nonlinear Poisson-like equation, along with an associated bilinear linearization form, both of which are required to use MCLite to produce a adaptive finite element approximation.

In the case of a nonlinear Poisson-like equation (PE) on finite domain $\Omega \subset \mathbb{R}^2$, the strong form is:

$$-\nabla \cdot (a(x)\nabla\hat{u}(x)) + b(x, u(x)) \;=\; 0 \;\text{ in } \Omega, \tag{3.1}$$

$$\hat{u}(x) \;=\; g(x) \;\text{ on } \partial\Omega, \tag{3.2}$$

where $\partial_D\Omega \equiv \partial\Omega$, i.e., the entire boundary of the domain is of Dirichlet type in this case (so that $\text{meas}_{d-1}(\partial_D\Omega) > 0$ holds). In this case, we denote space $H^1_{0,D}(\Omega)$ as simply $H^1_0(\Omega)$.

The corresponding weak formulation then follows from the previous discussion, where $a(x, u, \nabla u) = a(x)I_{2\times 2}$, $b(x, u, \nabla u) = b(x, u)$, and $c(x) = 0$, leading to:

$$\text{Find } u \in H^1_0(\Omega) \text{ such that } \; < F(u), v >= 0 \quad \forall v \in H^1_0(\Omega), \tag{3.3}$$

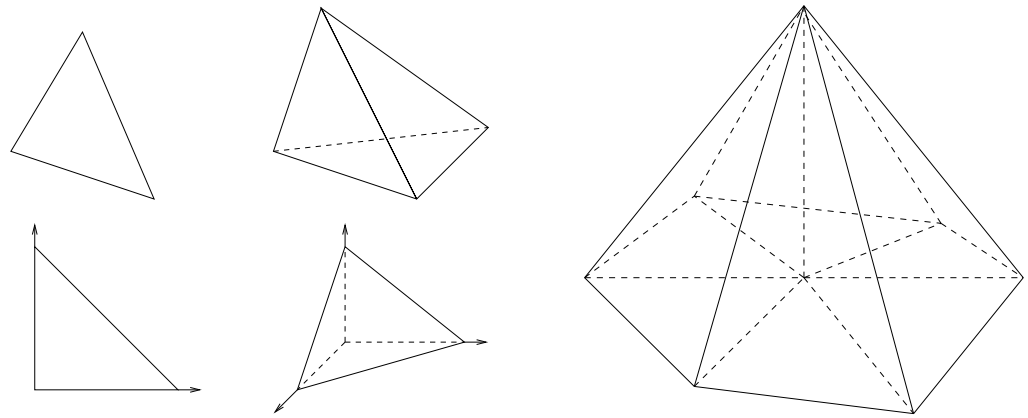

FIGURE 1. Reference and arbitrary 2- and 3-simplex elements, and a global (2D) basis function.

where

$$< F(u), v >= \int_\Omega a(x)(\nabla[u + u_D]) \cdot \nabla v + b(x, [u + u_D])v \; dx. \tag{3.4}$$

The bilinear linearization form $< DF(u)w, v >$ in the case of the PE is again produced as the Gateaux derivative of the nonlinear form $< F(u), v >$ above:

$$< DF(u)w, v >= \frac{d}{dt} < F(u + tw), v > \Big|_{t=0} = \int_\Omega \left(a(x)\nabla w \cdot \nabla v + \frac{\partial b}{\partial u}(x, u + u_D)wv \right) \; dx.$$

4. Multilevel adaptive finite element methods

4.1. **The finite element method for nonlinear elliptic equations.** A *Petrov-Galerkin* approximation of the solution to (2.8) is the solution to the following subspace problem:

$$\text{Find } u_h \in U_h \subset H^1_{0,D} \text{ such that } \; < F(u_h), v_h > = \; 0, \quad \forall v_h \in V_h \subset H^1_{0,D}, \tag{4.1}$$

for some chosen subspaces U_h and V_h of $H^1_{0,D}(\Omega)$. A *finite element* method is simply a Petrov-Galerkin approximation method in which the subspaces U_h and V_h are chosen to have the extremely simple form of continuous piecewise low-degree polynomials with local support, defined over a disjoint covering of the domain Ω by *elements*, typically simple polyhedra. For example, in the case of continuous piecewise linear polynomials defined over a disjoint covering with 2- or 3-simplices (cf. Figure 1), the basis functions are easily defined element-wise using the unit 2-simplex (triangle) and unit 3-simplex (tetrahedron) as follows:

$$\begin{aligned} \tilde{\phi}_0(\tilde{x}, \tilde{y}) &= 1 - \tilde{x} - \tilde{y} \\ \tilde{\phi}_1(\tilde{x}, \tilde{y}) &= \tilde{x} \\ \tilde{\phi}_2(\tilde{x}, \tilde{y}) &= \tilde{y} \end{aligned} \qquad \begin{aligned} \tilde{\phi}_0(\tilde{x}, \tilde{y}, \tilde{z}) &= 1 - \tilde{x} - \tilde{y} - \tilde{z} \\ \tilde{\phi}_1(\tilde{x}, \tilde{y}, \tilde{z}) &= \tilde{y} \\ \tilde{\phi}_2(\tilde{x}, \tilde{y}, \tilde{z}) &= \tilde{x} \\ \tilde{\phi}_3(\tilde{x}, \tilde{y}, \tilde{z}) &= \tilde{z} \end{aligned}$$

For a simplex mesh with n vertices, a set of n *global* basis functions $\{\phi_i(x, y, z), i = 1, \ldots, n\}$ are defined, as in the right-most picture in Figure 1, by combining the support regions around a given vertex, and extending the unit simplex basis functions to each arbitrary simplex using (affine) coordinate transformations.

The above basis functions clearly do not form any subspace of $\mathcal{C}^2(\Omega)$, the space of twice continuously differentiable functions on Ω, which is the natural function space in which to look for the solutions to second order elliptic equations written in the strong form, such as equation (2.1). This is due to the fact that they are discontinuous along simplex faces and simplex vertices in the disjoint simplex covering of Ω. However, one can show that in fact [20]:

$$V_h = \text{span}\{\phi_1, \ldots, \phi_n\} \subset H^1_{0,D}(\Omega), \; \Omega \subset \mathbb{R}^d,$$

so that these continuous, piecewise defined, low-order polynomial spaces do in fact form a subspace of the solution space to the weak formulation of the problems we are considering here. Taking then $U_h =$

$\text{span}\{\phi_1, \phi_2, \ldots, \phi_n\}$, $V_h = \text{span}\{\psi_1, \psi_2, \ldots, \psi_n\}$, equation (4.1) reduces to a set of n nonlinear algebraic relations (implicitly defined) for n coefficients $\{\alpha_k\}$ in the expansion

$$u_h = \sum_{k=1}^{n} \alpha_k \phi_k.$$

In particular, regardless of the complexity of the form $< F(u), v >$, as long as we can evaluate it for given arguments u and v, then we can evaluate the nonlinear discrete residual of the finite element approximation u_h as:

$$r_i =< F(\sum_{k=1}^{n} \alpha_k \phi_k), \psi_i >, \quad i = 1, \ldots, n.$$

Since the form $< F(u), v >$ involves an integral in this setting, if we employ quadrature then we can simply sample the integrand at quadrature points; this is a standard technique in finite element technology. Given the local support nature of the functions ϕ_k and ψ_i, all but a small constant number of terms in the sum $\sum_{k=1}^{n} \alpha_k \phi_k$ are zero at a particular spatial point in the domain, so that the residual r_i is inexpensive to evaluate when quadrature is employed. In many cases, the discrete polynomial "test" subspace is taken to be the same as the solution space, or $V_h \equiv U_h$, in which case the method is referred to as a Galerkin rather than a Petrov-Galerkin method.

The two primary issues in applying this approximation method are then:

(1) The approximation error $\|u - u_h\|_X$, for various norms X,
(2) The computational complexity of solving the n algebraic equations.

The first of these issues represents the core of finite element approximation theory, which itself rests on the results of classical approximation theory. Classical references to both topics include [20, 23, 21]. The second issue is addressed by the complexity theory of direct and iterative solution methods for sparse systems of linear and nonlinear algebraic equations, cf. [28, 42].

4.2. **Error estimation, adaptive simplex subdivision, and conformity.** *A priori* error analysis for the finite element method for addressing the first issue is now a well-understood subject [19, 20]. Much activity has recently been centered around *a posteriori* error estimation, and its use in adaptive mesh refinement algorithms [4, 3, 6, 43, 44, 45]. These estimators include weak and strong residual-based estimators [4, 3, 43], as well as estimators based on the solution of local problems [11, 13]. The challenge for a numerical method is to be as efficient as possible, and *a posteriori* estimates are a basic tool in deciding which parts of the solution require additional attention. While the majority of the work on *a posteriori* estimates has been for linear problems, nonlinear extensions are possible through linearization theorems (cf. [43, 44]). The solve-estimate-refine structure in MCLite (similar to that in MC) exploiting these *a posteriori* estimators is as follows:

Algorithm 4.1. *(Adaptive multilevel finite finite element approximation)*

- *While* $(\|u - u_h\|_X > TOL)$ *do:*
  (1) *Find* $u_h \in U_h$ *such that* $< F(u_h), v_h >= 0, \forall v_h \in V_h$
  (2) *Estimate* $\|u - u_h\|_X$ *over each element, set* $Q1 = Q2 = \emptyset$.
  (3) *Place simplices with large error in "refinement"* $Q1$
  (4) *Bisect all simplices in* $Q1$ *(removing them from* $Q1$*), and place any nonconforming simplices created in* $Q2$.
  (5) $Q1$ *is now empty; set* $Q1 = Q2$, $Q2 = \emptyset$.
  (6) *If* $Q1$ *is not empty, goto (4).*
- *end while*

The conformity loop (4)–(6), required to produce a globally "conforming" mesh (described below) at the end of a refinement step, is guaranteed to terminate in a finite number of steps (cf. [38, 39]), so that the refinements remain local. Element shape is crucial for approximation quality; the bisection procedure in step (4) is guaranteed to produce nondegenerate families if the longest edge is bisected is used in 2D [40, 41], and if marking or homogeneity methods are used in 3D [2, 15, 14, 32, 35, 37]. Whether longest edge bisection is nondegenerate in 3D apparently remains an open question. Figure 2 shows a single subdivision of a 2-simplex or a 3-simplex using either 4-section (first figure from left), 8-section (fourth figure from left), or bisection (third and sixth figures from left). The paired triangle in the 2-simplex case of Figure 2 illustrates

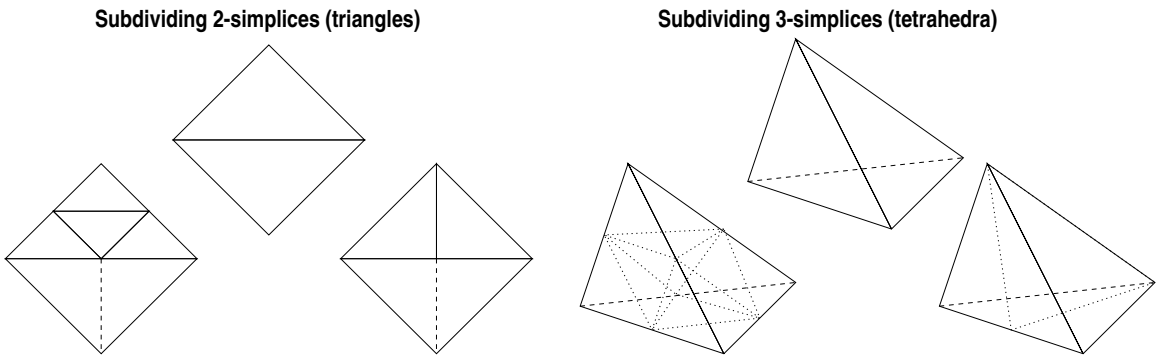

FIGURE 2. Refinement of 2- and 3-simplices using 4-section, 8-section, and bisection.

the nature of conformity and its violation during refinement. A globally conforming simplex mesh is defined as a collection of simplices which meet only at vertices and faces; for example, removing the dotted bisection in the third figure from the left in Figure 2 produces a non-conforming mesh. Non-conforming simplex meshes create several theoretical as well as practical implementation difficulties, and the algorithms in MCLite (as well as those in MC [29] and PLTMG [6], and similar simplex-based adaptive codes [29, 37, 17, 18, 16]) enforce conformity using the above finite queue swapping strategy or a similar approach.

4.3. **Inexact-Newton methods and linear multilevel methods.** Addressing the complexity of Step 1 of the algorithm above, Newton methods are often the most effective:

Algorithm 4.2. *(Damped-inexact-Newton)*

- *Given an initial* u
- *While* $(| < F(u), v > | > TOL \ \forall v)$ *do:*
  (1) *Find* w *such that* $< DF(u)w, v >= - < F(u), v > +r, \ \forall v$
  (2) *Set* $u = u + \alpha w$
- *end while*

The bilinear form $< DF(u)w, v >$ in the algorithm above is the (Gateaux) linearization of the nonlinear form $< F(u), v >$, defined earlier. This form is easily computed from most nonlinear forms $< F(u), v >$ which arise from second order nonlinear elliptic problems. The possibly nonzero "residual" term r is to allow for inexactness in the Jacobian solve for efficiency, which is quite effective in many cases; cf. [10, 22, 24]. The parameter α brings robustness to the algorithm [8, 9, 24]. If folds or bifurcations are present, then the iteration is modified to incorporate path-following [7, 30].

As was the case for the nonlinear residual $< F(\cdot), \cdot >$, the matrix representing the bilinear form in the Newton iteration is easily assembled, regardless of the complexity of the bilinear form $< DF(\cdot)\cdot, \cdot >$. In particular, the matrix equation for $w = \sum_{j=1}^{n} \beta_j \phi_j$ has the form:

$$AU = F,$$

where

$$A_{ij} =< DF(\sum_{k=1}^{n} \alpha_k \phi_k)\phi_j, \psi_i >, \qquad F_i =< F(\sum_{k=1}^{n} \alpha_k \phi_k), \psi_i >, \qquad U_i = \beta_i.$$

As long as the integral-based forms $< F(\cdot), \cdot >$ and $< DF(\cdot)\cdot, \cdot >$ can be evaluated at individual points in the domain, then quadrature can be used to build the Newton equations, regardless of the complexity of the forms. This is one of the most powerful features of the finite element method, and is exploited to an extreme in the code MCLite and MC (see [29]).

It can be shown that the Newton iteration above is dominated by the computational complexity of solving the n linear algebraic equations in each iteration; cf. [10, 26]. We use MATLAB's builtin sparse direct solver in MCLite for the solution of these equations, which is know to have complexity of $O(n^{1.5})$ for model problems in two dimensions. However, the complexity of sparse direct methods decays to $O(n^2)$ in three dimensions, and as a result we employ algebraic multilevel methods in the MC implementation (cf. [29]).

5. The computer program MCLite

MCLite is an adaptive multilevel finite element MATLAB code developed by M. Holst over several years at Caltech and UC San Diego. It is a two-dimensional MATLAB prototype of the two- and three-dimensional ANSI-C code MC [29, 5], also written by M. Holst. MCLite is designed to produce provably accurate numerical solutions to two-dimensional scalar nonlinear elliptic equations in an efficient way. MCLite employs *a posteriori* error estimation, adaptive simplex subdivision, global inexact-Newton methods, and numerical continuation methods, for the highly accurate numerical solution of this class of problems. The MC ANSI-C implementation extends the MCLite MATLAB capabilities to both two- and three-dimensional nonlinear elliptic systems of tensor equations on two- and three-manifolds, by employing unstructured algebraic multilevel methods for the linear systems which arise; cf. [29].

5.1. **The overall design of MCLite.** The MCLite code is an implementation of Algorithm 4.1, employing Algorithm 4.2 for nonlinear elliptic equations that arise in Step 1 of Algorithm 4.1. The linear Newton equations in each iteration of Algorithm 4.2 are solved with MATLAB's builtin sparse direct solvers. The Newton equations are supplemented with a continuation algorithm when necessary. Several of the features of MCLite, which are inherited from MC, are somewhat unusual, allowing for the treatment of very general nonlinear elliptic operators and general domains. In particular, some of these features are:

- *Abstraction of the elliptic equation*: The elliptic equation is defined only through a nonlinear weak form over the domain, along with an associated linearization form, also defined everywhere on the domain. (precisely the forms $< F(u), v >$ and $< DF(u)w, v >$ in the discussions above).
- *Abstraction of the domain manifold*: The domain manifold is specified by giving a polyhedral representation of the topology, along with a set of coordinate labels. MCLite works primarily with the topology of the domain (i.e., the connectivity of the polyhedral representation). The geometry of the domain manifold is provided primarily through the form definitions. (MC carries this a bit further, allowing for the use of 2- and 3-manifolds with multiple coordinate systems as PDE domains.)
- *Dimension-independence*: While MCLite is a two-dimensional implementation, all of the algorithms extend immediately to three (and higher) dimensions. To achieve this dimension-independence, MCLite employs the simplex as its fundamental geometrical object for defining finite element bases. (This allows MCLite to be used as a prototype code for trying things out that will later be implemented in the ANSI-C code MC.)

The abstract weak form approach to defining the problem allows for the complete definition of a problem in MCLite by writing only a few lines of MATLAB to define the two weak forms. Changing to a different problem involves providing only a different definition of the forms and a different domain description.

5.2. **Topology and geometry representation in MCLite.** A datastructure referred to as the *ringed-vertex* (cf. [29]) is used to represent meshes of d-simplices of arbitrary topology. This datastructure, illustrated in Figure 3, is similar to the winged-edge, quad-edge, and edge-facet datastructures commonly used in the computational geometry community for representing 2-manifolds [36], but it can be used more generally to represent arbitrary d-manifolds, $d = 2, 3, \ldots$. It maintains a mesh of d-simplices with near minimal storage, yet for shape-regular (non-degenerate) meshes, it provides $O(1)$-time access to all information necessary for refinement, un-refinement, and discretization of an elliptic operator. The ringed-vertex datastructure also allows for dimension-independent implementations of mesh refinement and mesh manipulation, with one implementation covering arbitrary dimension d. An interesting feature of this datastructure is that the structures used for vertices, simplices, and edges are all of fixed size, so that a fast array-based implementation is possible, as opposed to a less-efficient list-based approach commonly taken for finite element implementations on unstructured meshes. A detailed description of the ringed-vertex datastructure, along with a complexity analysis of various traversal algorithms, can be found in [29].

Since MCLite is based entirely on the d-simplex, for adaptive refinement it employs simplex bisection, using one of the simplex bisection strategies outlined earlier. Bisection is first used to refine an initial subset of the simplices in the mesh (selected according to some error estimates, discussed below), and then a closure algorithm is performed in which bisection is used recursively on any non-conforming simplices, until a conforming mesh is obtained (outlined above). If it is necessary to improve element shape (which effects finite element approximation quality), MCLite attempts to optimize the following simplex shape measure

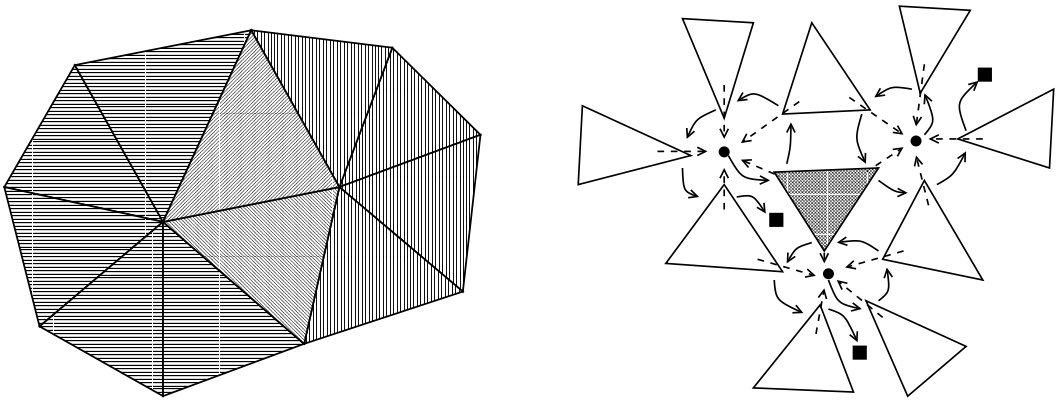

FIGURE 3. Polyhedral mesh representation. The figure on the left shows the overlapping support regions of two vertex-based global basis functions. The polyhedral mesh topology is represented by MCLite using a *ringed vertex* datastructure illustrated in the figure on the right; the topology primitives are vertices and d-simplices. (Edges are temporarily created during subdivision, but are then destroyed; a similar ring datastructure is used to represent the edge topology.)

function for a given d-simplex s, in an iterative fashion, similar to the approach taken in [12]:

$$\eta(s,d) = \frac{2^{2(1-\frac{1}{d})} 3^{\frac{d-1}{2}} |s|^{\frac{2}{d}}}{\sum_{0 \leq i < j \leq d} |e_{ij}|^2}$$

The quantity $|s|$ represents the (possibly negative) volume of the d-simplex s, and $|e_{ij}|$ represents the length of the edge that connects vertex i to vertex j in the simplex. For $d = 2$, this is the shape-measure used in [12], with a slightly different normalization. For $d = 3$, this is the shape-measure given in [31], again with a slightly different normalization.

5.3. **Discretization, error estimation, and adaptivity in MCLite.** MCLite uses piecewise linear elements to discretize the elliptic equation, The equations is the solved using MATLAB's builtin sparse direct solver, and *a posteriori* error estimates are computed from the discrete solution to drive adaptive mesh refinement. The idea of adaptive error control in finite element methods is to estimate the behavior of the actual solution to the problem using only a previously computed numerical solution, and then use the estimate to build an improved numerical solution by upping the polynomial order (p-refinement) or refining the mesh (h-refinement) where appropriate. Note that this approach to adapting the mesh (or polynomial order) to the local solution behavior affects not only approximation quality, but also solution complexity: if a target solution accuracy can be obtained with fewer mesh points by their judicious placement in the domain, the cost of solving the discrete equations is reduced (sometimes dramatically) because the number of unknowns is reduced (again, sometimes dramatically). Generally speaking, if an elliptic equation has a solution with local singular behavior, such as would result from the presence point loads (delta functions) in the forcing function (right hand side of equation (2.1)), then adaptive methods tend to give dramatic improvements over non-adaptive methods.

There are several approaches to adaptive error control, although the following two approaches based on *a posteriori* error estimation are usually the most effective (and are in fact equivalent in a certain sense [43]):

(1) Residual estimates (strong and weak) [33, 34, 43, 44]
(2) Solution of local problems (Dirichlet or Neumann) [13, 11]

Most existing work on *a posteriori* estimates has been for linear problems; nonlinear extensions through linearization. For example, consider the nonlinear problem:

$$F(u) = 0,\ F \in C^1(X, Y^*),\ X, Y \text{ Banach spaces},$$

and a discretization:

$$F_h(u_h) = 0,\ F_h \in C^0(X_h, Y_h^*),\ X_h \subset X,\ Y_h \subset Y.$$

Nonlinear residuals $F(u_h)$ can be used to estimate the error $\|u - u_h\|_X$, using linearization theorems; here is an example of such a theorem due to Verürth [43].

Theorem 5.1. *Let $u \in X$ be a regular solution of $F(u) = 0$, so that $DF(u)$ is a linear homeomorphism of X onto Y^*. Assume DF is Lipschitz continuous at u, so that there exists R_0 such that*

$$\gamma = \sup_{u_h \in B(u,R_0)} \frac{\|DF(u) - DF(u_h)\|_{\mathcal{L}(X,Y^*)}}{\|u - u_h\|_X} < \infty.$$

Let $R = \min\{R_0, \gamma^{-1}\|DF(u)^{-1}\|_{\mathcal{L}(Y^*,X)}, 2\gamma^{-1}\|DF(u)\|_{\mathcal{L}(X,Y^*)}\}$ Then, for all $u_h \in B(u,R)$,

$$\left[\frac{1}{2}\|DF(u)\|^{-1}_{\mathcal{L}(X,Y^*)}\right] \cdot \|F(u_h)\|_{Y^*} \leq \|u - u_h\|_X \leq \left[2\|DF(u)^{-1}\|_{\mathcal{L}(Y^*,X)}\right] \cdot \|F(u_h)\|_{Y^*}$$

The effect of linearization is swept under the rug somewhat by the choice of R sufficiently small. However, since we typically construct highly refined meshes where needed, such local linear estimates are reasonable. Note that $\|F(u_h)\|_{Y^*}$ can be estimated in several ways, including

(1) Approximation by $\|F_h(u_h)\|_{Y^*}$
(2) Solution of local problems

The two approaches are often equivalent up to constants. For reasons of efficiency in the case of elliptic systems in spatial dimension three, we employ the first approach in MC, referred to as estimation by strong residuals. In MCLite, the error estimator is user-specified; the user is given all of the information about an element and the current discrete solution in the element, and is asked to produce an error estimate for the element. This can be as simple as computing the average of the jumps in the normal derivatives of the solution gradient across the edges shared with neighboring elements, or the complete calculation of a bound on the strong form of the residual [43]. In particular, one employs the linearization theorem above, together with some derived (but computable) upper and lower bounds on the nonlinear residual $\|F(u_h)\|_{Y^*}$ given by the following pair of inequalities:

$$A \leq \|F(u_h)\|_{Y^*} \leq B.$$

There exact forms of A and B, and their efficient calculation, is quite problem dependent in the nonlinear case; cf. [43].

5.4. **Solution of of linear and nonlinear systems with MCLite.** When a system of nonlinear finite element equations must be solved in MCLite, the global inexact-Newton Algorithm 4.2 is employed, where the linearization systems are solved by MATLAB's builtin sparse direct solver. When necessary, the Newton procedure in Algorithm 4.2 is supplemented with a user-defined normalization equation for performing an augmented system continuation algorithm. Coupled with the superlinear convergence properties of the outer inexact Newton iteration in Algorithm 4.2, this leads to an overall complexity of $O(N^{1.5})$ for the solution of the discrete nonlinear problems in Step 1 of Algorithm 4.1. Combining this low-complexity solver with the judicious placement of unknowns only where needed due to the error estimation in Step 2 and the subdivision algorithm in Steps 3-6 of Algorithm 4.1, leads to an effective low-complexity approximation technique for solving this general class of elliptic problems.

6. Availability of MCLite

MCLite and various supporting tools are available under the GNU copyleft license, and may be found at the following website:

`http://www.scicomp.ucsd.edu/~mholst/codes/mclite/`

Appendix: Actually Using MCLite

Here are is a brief description of all routines in the package; this information is also contained in the `README` file distributed with the package.

```
% documentation files
file README        # THIS FILE -- description of the package
file COPYING       # the GNU license which governs package distribution
file QUICKSTART    # if you have no patience
```

```
% primary controlling routines
file go.m          # main MCLite driver program
file assem.m       # assemble the discrete system
file newton.m      # solve the discrete system with newton's method
file mastr.m       # get master element information
file read.m        # read in a mesh from a standard "MCSF" mesh file
file write.m       # write out a mesh to a standard "MCSF" mesh file
file writemcs.m    # low-level routine to write an "MCSF" mesh file
file writemce.m    # low-level routine to write an "MCEF" mesh file
file writeoff.m    # write out mesh to standard "OFF" file or socket
file writeoffs.m   # write out solution to standard "OFF" file or socket
file zedgmesh.m    # builds an MCEF mesh by treating a matrix as the graph
file mcin.m        # a sample "MCSF" mesh file (the unit square)
file mcinTOR.m     # a sample "MCSF" mesh file (a torus)
file mcout.m       # a sample "MCSF" mesh file (another torus)
file mcout.off     # a sample "OFF" mesh file

% high-level error estimation and refinement routines
file mark.m        # tag simplices for refinement (e.g., error estimation)
file refin.m       # refine a set of marked simplices until conformity
file refinbis.m    # one pass through simplices, bisecting marked ones
file refinqud.m    # one pass through simplices, quadrasecting marked ones

% low-level simplex ring manipulation routines
file bldsring.m    # build simplex rings for all simplices in the mesh
file kilring.m     # destroy simplex rings for all simplices in the mesh
file chkmesh.m     # extensive consistency/conformity check of entire mesh
file addsring.m    # add a single simplex to all of its simplex rings
file delsring.m    # delete a single simplex from all of its rings
file sinring.m     # is a particular simplex in a particular simplex ring
file addering.m    # add a single edge to all of its edge rings
file getedge.m     # get the unique edge (if exists) between two vertices
file getnabor.m    # get the unique simplex (if exists) sharing a face
file longedge.m    # determine the longest edge of a single simplex
file vneum.m       # yes/no answer about whether tri/edge/vert is neumann
file vdiri.m       # yes/no answer about whether tri/edge/vert is dirichlet

% misc untilities and plotting tools
file assert.m      # an assertion macro for verifying correctess/debugging
file mypaus.m      # pause until the return key is pressed
file draw1.m       # draw a mesh of vertices and simplices (mode 1)
file draw2.m       # draw a mesh of vertices and simplices (mode 2)
file draw3.m       # draw a mesh of vertices and simplices (mode 3)
file drawf.m       # draw a function over a mesh of vertices and simplices
file u.m           # move the plot up
file d.m           # move the plot down
file r.m           # move the plot right
file l.m           # move the plot left

% problem specification (user-defined)
file fu_v.m        # nonlinear weak form defining the pde
file dfu_wv.m      # linearization bilinear form for the pde
file u_d.m         # dirichlet boundary function
file u_t.m         # optional analytical solution for testing purposes
file edgsplit.m    # user-defined procedure to bisect edges in the mesh
```

There is usually a very descriptive header block in each MATLAB routine; one can view this documentation for the routine by either editing the MATLAB routine, or by typing e.g. `help read` at the MATLAB

prompt to view the documentation for the routine `read.m`. This documentation will be your best guide to understanding and using the MCLite package. I stress again: read the documentation in the header blocks of the routines, and also read the documentation in the code itself; it is quite well-documented throughout (that is the only way I can write code and expect to understand it the next day!)

The most confusing thing about the MCLite package is usually the ringed vertex datastructure that is used to represent the simplex mesh. By reading the documentation in the file `read.m`, you will understand exactly what the basic datastructure is. The simplex rings are simply linked lists of simplices that begin at vertices; from any vertex, you can walk this list to find all simplices (triangles in MCLite) which use that vertex. By using this single structure, you can assemble information such as which simplices are edge neighbors, and so forth. Similarly, the edge rings are linked lists of edges which begin at vertices; from any vertex, you can find all edges which use that vertex. By using this single structure, you can assemble information such as which vertices share edges, and so forth. The routines `getnabor.m` and `sinring.m` do some of these things; again, start reading the documentation in the routines and you will begin to understand the overall structure.

The routines in the "`low-level simplex ring manipulation routines`" section above simply build, destroy, traverse, and manipulate the ringed vertex datastructure (the simplex and edge rings around a vertex). The actual ring datastructures are built by the routines `bldsring.m`, `addsring.m`, and `addering.m`. The routine `bldsring.m` is called just before a mesh is refined in the routine `refin.m`; `bldsring.m` simply calls `addsring.m` for each simplex, which adds each simplex to the simplex rings that start at each of its vertices. The routine `addering.m` adds an edge to the edge rings that start at each of its two vertices. There are several advantages to using the ringed vertex datastructure; the primary one as far as MCLite is concerned is that all vertices, edges, and simplices have *fixed size*, so that an array-based implementation in MATLAB is possible. (This also makes for a faster ANSI-C implementation in MC, compared to linked lists of variable-sized datastructures.)

Here are a few additional guidelines and pointers to help you get started. The user tells MCLite about the particular PDE to solve by specifying the nonlinear weak form in the file `fu_v.m`, along with its linearization weak form `dfu_wv.m`. (We will look more closely at these two files below.) The user also specifies the Dirichlet boundary condition function as the function `u_d.m`. The function `u_t.m` is a function that is provided as a slot for an analytical solution to the PDE for testing purposes, if available (otherwise `u_t.m` can simply return with no action).

The specification of the domain over which to solve the PDE adaptively is effected by providing a simplex mesh input file `mcin.m` in MCSF format, along with a single routine `edgsplit.m` which tells MCLite how to bisect edges in the mesh when doing refinement. The exact specification of the MCSF format is described in detail in the header to the mesh read routine `read.m`. The `edgsplit.m` file can be simply the usual bisection of a line in the plane (the average of the two input points), or it can be more complicated to reflect the correct refinement of boundary edges, or to allow for the underlying domain to be a 2-manifold (this is done in a more sophisticated manor in MC to handle tensor equations correctly; cf. [29]).

MCLite is used by beginning a MATLAB session, and then executing the script `go.m`. This script is a simple driver that first reads in a mesh in MCSF format (the simplex mesh input file format that MCLite and MC share), and then goes into the discretize/solve/estimate/refine loop in Algorithm 4.1. The script can be completely controlled by editing it and changing the various parameters that appear in the "`controlling parameters`" section of the driver. Along with the PDE and mesh specification routines described above, the MCLite package can be used simply by editing and running `go.m`.

The `go.m` file is listed below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Prototype:
%
%    None (script)
%
% Purpose:
%
%    Driver script for MCLite package.
%
% Author:   Michael Holst
```

```
%

%%% some i/o

   clc;
   % clf;
   fprintf('\n');
   fprintf('------------------------------------------------------------\n');
   fprintf('MCLite version 1.0, Copyright (C) 1998 Michael Holst\n');
   fprintf('This is free software, and you are welcome to redistribute it\n');
   fprintf('under certain conditions; see the file "COPYING" for details.\n');
   fprintf('Note that MCLite comes with ABSOLUTELY NO WARRANTY.\n');
   fprintf('\n');
   fprintf('For MCLite documentation:  In MATLAB, type "help filename".\n');
   fprintf('------------------------------------------------------------\n');
   fprintf('\n');

   fprintf('MAIN: STARTING: MCLite (MC-Lite) Code\n');

%%% controlling parameters

   levels      = 23; %%% num of refinements plus one
   refkey      = 1;  %%% refinement type (0=bisection,1=quadrasection)
   estkey      = 1;  %%% estimator (0=mark all,1=geometric,2=a posteriori)
   solkey      = 1;  %%% solve key (0=no, 1=yes)
   chkkey      = 0;  %%% conformity check the mesh (0=no,1=yes--check is slow)

   iplotMan    = 1;  %%% MATLAB plot of manifold mesh (0=none,1=manifold)
   iplotMesh   = 1;  %%% MATLAB plot of mesh (0=none,1=mesh)
   igraphA     = 1;  %%% MATLAB graph of matrix (0=none,1=graph)
   ispyA       = 1;  %%% MATLAB spy of matrix (0=none,1=spy)
   iplotSol    = 1;  %%% MATLAB plot of solution (0=none,1=soln)
   igenPS      = 1;  %%% MATLAB generate postscript (0=no,1=yes)

   gv          = 0;  %%% MCgl or Geomview plot of manifold mesh (0=none,1=yes)
   gvs         = 0;  %%% MCgl or Geomview plot of solution mesh (0=none,1=yes)
   socktype    = 0;  %%% UNIX domain or INET sockets (0=UNIX domain, 1=INET)

%%% construct operators for the nested sequence of spaces

   for level = 1 : levels
      fprintf('MAIN: -------\n');
      fprintf('MAIN: LEVEL %g\n',level);
      fprintf('MAIN: -------\n');

      if (level == 1)
         %%% create the initial coarse mesh
         fprintf('MAIN: defining the mesh..');
         t0 = clock;
         [VERT,SIMP] = read(0);
         the_time = etime(clock,t0);
         [N,eight]     = size(VERT);
         [L,seventeen] = size(SIMP);
         fprintf('..done.  [N=%g,L=%g]  [time=%g]\n',N,L,the_time);
      else
         %%% error estimate and the mark mesh (some uniform refinements first)
         fprintf('MAIN: marking  the mesh..');
         t0 = clock;
         [SIMP,QUE]    = mark(VERT,SIMP,estkey);
```

```
   the_time = etime(clock,t0);
   [N,eight]     = size(VERT);
   [L,seventeen] = size(SIMP);
   fprintf('..done.  [N=%g,L=%g]  [time=%g]\n',N,L,the_time);

   %%% adaptively refine the mesh based on the marking
   fprintf('MAIN: refining the mesh..');
   t0 = clock;
   [VERT,SIMP]   = refin(VERT,SIMP,QUE,refkey);
   the_time = etime(clock,t0);
   [N,eight]     = size(VERT);
   [L,seventeen] = size(SIMP);
   fprintf('..done.  [N=%g,L=%g]  [time=%g]\n',N,L,the_time);
end;

%%% check the mesh for "correctness"
if (chkkey > 0)
   chkmesh(VERT,SIMP);
end

%%% plot the mesh
if (gv > 0)
   writeoff(socktype,2,VERT,SIMP);
   mypaus;
end;
if (iplotMan > 0)
   fprintf('MAIN: plotting the manifold..');
   draw1(VERT,SIMP);
   fprintf('..done.  [N=%g,L=%g]\n',N,L);
   mypaus;
   draw2(VERT,SIMP);
   fprintf('..done.  [N=%g,L=%g]\n',N,L);
   mypaus;
end;
if (iplotMesh > 0)
   fprintf('MAIN: plotting the mesh..');
   draw3(VERT,SIMP);
   if (igenPS > 0)
      print -depsc mesh.eps
   end
   fprintf('..done.  [N=%g,L=%g]\n',N,L);
   mypaus;
end;

%%% solve discrete system?
if (solkey > 0)

    %%% solve discrete system
    fprintf('MAIN: nonlinear solve starting.  [N=%g,L=%g]\n',N,L);
    t0 = clock;
    [U,Ug,A,F]=newton(VERT,SIMP);
    the_time = etime(clock,t0);
    fprintf('MAIN: nonlinear solve finished.  [time=%g]\n',the_time);

    %%% discretization error test
    U_t = zeros(N,1);
    U_t(1:N,1) = u_t(VERT(1:N,1), VERT(1:N,2));
    error = norm(U_t - (U+Ug)) / norm(U_t);
    fprintf('MAIN: discretization error is = %g\n',error);
```

```
      %%% graph the matrix
      if (igraphA > 0)
         fprintf('MAIN: graphing matrix..');
         gplot(A,VERT,'b');
         if (igenPS > 0)
            print -depsc graph.eps
         end
         fprintf('..done.  [N=%g,L=%g]\n',N,L);
         mypaus;
      end;

      %%% show matrix structure
      if (ispyA > 0)
         fprintf('MAIN: showing nonzero structure..');
         clf; hold off;
         spy(A);
         if (igenPS > 0)
            print -depsc spy.eps
         end
         fprintf('..done.  [N=%g,L=%g]\n',N,L);
         mypaus;
      end;

      %%% plot the solution
      if (gvs > 0)
         writeoffs(socktype,2,VERT,SIMP,(U+Ug));
         mypaus;
      end;
      if (iplotSol > 0)
         fprintf('MAIN: plotting FEM solution..');
         drawf(VERT,SIMP,(U+Ug));
         view(30,30);
         if (igenPS > 0)
            print -depsc sol.eps
         end
         fprintf('..done.  [N=%g,L=%g]\n',N,L);
         mypaus;
      end;

   end; %%% solve discrete system

end; %%% solve-estimate-refine loop
```

Finally, regarding the weak form routines `f_uv.m` and `dfu_wv.m`, with which you specify completely your PDE (along with the Dirichlet function `u_d.m`): these two routines must return the value of the *integrand* in the weak form integral, evaluated at the list of points that are passed into the routines. This is because MCLite assembles the nonlinear residual and the linearized stiffness matrix using (Gaussian) quadrature. MCLite assembles all of the elements at once, so that what is passed to the three routines `f_uv.m`, `dfu_wv.m`, and `u_d.m` is basically the list of all of the quadratures points required to approximate the integrals need to assemble all of the element matrices at once.

For a clear example of how to specify these three routines, consider the Bratu problem (for constant $a \in \mathbb{R}$):

$$\begin{aligned} -\nabla \cdot (a \nabla u) + \lambda e^u &= f, \quad \text{in } \Omega = [0,1] \times [0,1], \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned}$$

The weak form $< F(u), v >$ is:

$$< F(u), v >= \int_\Omega (a\nabla u \cdot \nabla v + \lambda e^u v - fv)\ dv,$$

and the bilinear linearization form $< DF(u)w, v >$ is:

$$< DF(u)w, v >= \int_\Omega (a\nabla w \cdot \nabla v + \lambda e^u wv)\ dv.$$

If we choose $f(x,y) = 2\pi^2 a \sin(\pi x)\sin(\pi y) + \lambda e^{\sin(\pi x)\sin(\pi y)}$, then the analytical solution to this problem is $u(x,y) = \sin(\pi x)\sin(\pi y)$. For this problem, The correct specification of the two forms, the Dirichlet function, and the analytical solution (which is available in this particular case), are as follows:

```
function [thetam] = fu_v(evalKey, parm, L, xm,ym, u,ux,uy, pr,prx,pry);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Prototype:
%
%     [thetam] = fu_v(evalKey, parm, L, xm,ym, u,ux,uy, pr,prx,pry);
%
% Purpose:
%
%     Evaluate the nonlinear form by providing the value of the quantity
%     under the integral sign, at a particular quadrature point.
%
%     Because we do this for all of the elements at one time to make things
%     reasonably fast, we pass some arrays around.
%
% Input:
%
%     evalKey    ==> area or surface integral
%     L          ==> number of elements to do all at once
%     xm(1:L,1)  ==> x-coordinates of quadrature points, one per element
%     ym(1:L,1)  ==> y-coordinates of quadrature points, one per element
%     u(1:L,1)   ==> u, evaluated on master element
%     ux(1:L,1)  ==> partial w.r.t. x of u, one per element
%     uy(1:L,1)  ==> partial w.r.t. x of u, one per element
%     pr         ==> phi(r), evaluated on master element
%     prx(1:L,1) ==> partial w.r.t. x of phi(r), one per element
%     pry(1:L,1) ==> partial w.r.t. x of phi(r), one per element
%
% Output:
%
%     thetam     ==> the integrand for the area or surface integral,
%                    not including the jacobian of transformation
%
% Author:   Michael Holst
%

   lambda = parm(1);

   %%% area integral
   if (evalKey == 0)

      %%% coefficients
      atilde = ones(L,1);   % the first order coefficient function

      %%% source term chosen to give (one) analytical solution
      utilde(1:L,1) = sin(pi*xm) .* sin(pi*ym);
      ftilde(1:L,1) = (2*pi*pi) * atilde(1:L,1) .* utilde(1:L,1) ...
                    + lambda * exp(utilde(1:L,1));
```

```
      %%% nonlinear residual of the given FEM function "u"
      thetam(1:L,1) = ...
         atilde(1:L,1) .* (ux(1:L,1).*prx(1:L,1)+uy(1:L,1).*pry(1:L,1)) ...
       + (lambda * exp(u(1:L,1)) - ftilde(1:L,1)) * pr;

   %%% edge integral (for natural boundary condition)
   else if (evalKey == 1)

      %%% apply appropriate condition based on which boundary we are on
      thetam = zeros(L,1);
      for k=1:L
         if (xm(k) == 0)
             thetam(k) =  pi*cos(pi*xm(k))*sin(pi*ym(k))*pr;
         elseif (xm(k) == 1)
             thetam(k) = -pi*cos(pi*xm(k))*sin(pi*ym(k))*pr;
         elseif (ym(k) == 0)
             thetam(k) =  pi*sin(pi*xm(k))*cos(pi*ym(k))*pr;
         elseif (ym(k) == 1)
             thetam(k) = -pi*sin(pi*xm(k))*cos(pi*ym(k))*pr;
         else
             thetam(k) = 0;
         end
      end

   %%% error
   else
      assert( 0, 'fu_v 1' );
   end

end
function [thetam] = dfu_wv(evalKey, parm, L, xm,ym, u,ux,uy, ...
   ps,psx,psy, pr,prx,pry);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Prototype:
%
%     [thetam] = dfu_wv(evalKey, parm, L, xm,ym, u,ux,uy, ...
%        ps,psx,psy, pr,prx,pry);
%
% Purpose:
%
%     Evaluate the bilinear form by providing the value of the quantity
%     under the integral sign, at a particular quadrature point.
%
%     Because we do this for all of the elements at one time to make things
%     reasonably fast, we pass some arrays around.
%
% Input:
%
%     evalKey    ==> area or surface integral
%     L          ==> number of elements to do all at once
%     xm(1:L,1)  ==> x-coordinates of quadrature points, one per element
%     ym(1:L,1)  ==> y-coordinates of quadrature points, one per element
%     u(1:L,1)   ==> u, evaluated on master element
%     ux(1:L,1)  ==> partial w.r.t. x of u, one per element
%     uy(1:L,1)  ==> partial w.r.t. x of u, one per element
%     ps         ==> phi(s), evaluated on master element
%     psx(1:L,1) ==> partial w.r.t. x of phi(s), one per element
```

```
%     psy(1:L,1) ==> partial w.r.t. x of phi(s), one per element
%     pr         ==> phi(r), evaluated on master element
%     prx(1:L,1) ==> partial w.r.t. x of phi(r), one per element
%     pry(1:L,1) ==> partial w.r.t. x of phi(r), one per element
%
% Output:
%
%     thetam     ==> the integrand for the area or surface integral,
%                    not including the jacobian of transformation
%
% Author:   Michael Holst
%

   lambda = parm(1);

   %%% area integral
   if (evalKey == 0)

      %%% coefficients
      atilde = ones(L,1);   % the first order coefficient function

      %%% linearized operator evaluated at the FEM function "u"
      thetam(1:L,1) = ...
          atilde(1:L,1) .* (psx(1:L,1).*prx(1:L,1)+psy(1:L,1).*pry(1:L,1)) ...
        + lambda * exp(u(1:L,1)) * ps * pr;

   %%% edge integral (for natural boundary condition)
   else if (evalKey == 1)
      thetam(1:L,1) = zeros(L,1);

   %%% error
   else
      assert( 0, 'dfu_wv 1' );
   end

end


function [gxy] = u_d(x,y);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Prototype:
%
%    [gxy] = u_d(x,y);
%
% Purpose:
%
%    Essential boundary condition function.
%
% Author:   Michael Holst
%

   [L,one] = size(x);
   gxy = zeros(L,1);
   gxy = u_t(x,y);


function [uxy] = u_t(x,y);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% Prototype:
%
%    [uxy] = u_t(x,y);
%
% Purpose:
%
%    True solution (for testing).
%
% Author:   Michael Holst
%

   [L,one] = size(x);
   uxy = zeros(L,1);
   uxy = sin(pi*x) .* sin(pi*y);
```

To control the adaptivity in MCLite, the user provides an error estimator routine called `mark.m`. Given all of the information about an element and the current discrete solution in the element, this routine decides whether or not the element should be refined in order to reduce the element-wise error. The default `mark.m` routine provided with MCLite marks elements for refinement based on a simple geometric condition (if an element crosses the unit circle, then it is marked for refinement). Placing the error estimator in "user-land" allows for quite a bit of flexibility for trying out various error indicators.

Let's look at the behavior of MCLite when given the Bratu problem above, over the unit square, and given the simple geometric element marker as an error estimator. We will look at the I/O produced by MCLite while it is running, and then will show the graphical output such as the refined meshes, the graph of the linearized stiffness matrix, the nonzero structure of the matrix, and the plots of the solutions over the two-dimensional domain. After looking at this example, you should be ready to use MCLite!

```
>> go

------------------------------------------------------------
MCLite version 1.0, Copyright (C) 1998 Michael Holst
This is free software, and you are welcome to redistribute it
under certain conditions; see the file "COPYING" for details.
Note that MCLite comes with ABSOLUTELY NO WARRANTY.

For MCLite documentation:  In MATLAB, type "help filename".
------------------------------------------------------------

MAIN: STARTING: MCLite (MC-Lite) Code
MAIN: -------
MAIN: LEVEL 1
MAIN: -------
MAIN: defining the mesh....done.  [N=16,L=18]  [time=0.062056]
MAIN: plotting the manifold....done.  [N=16,L=18]
 --- press return ---
..done.  [N=16,L=18]
 --- press return ---
MAIN: plotting the mesh....done.  [N=16,L=18]
 --- press return ---
MAIN: nonlinear solve starting.  [N=16,L=18]
NEWTON: iter = 0, ||F(u)|| = 2.54987
NEWTON: iter = 1, ||F(u)|| = 6.67866e-16
MAIN: nonlinear solve finished.  [time=0.893346]
MAIN: discretization error is = 0.151521
MAIN: graphing matrix....done.  [N=16,L=18]
 --- press return ---
MAIN: showing nonzero structure....done.  [N=16,L=18]
 --- press return ---
MAIN: plotting FEM solution....done.  [N=16,L=18]
```

```
 --- press return ---
MAIN: -------
MAIN: LEVEL 2
MAIN: -------
MAIN: marking  the mesh....done.  [N=16,L=18]  [time=0.014956]
MAIN: refining the mesh....done.  [N=19,L=22]  [time=0.153644]
MAIN: plotting the manifold....done.  [N=19,L=22]
 --- press return ---
.
.
.
MAIN: -------
MAIN: LEVEL 6
MAIN: -------
MAIN: marking  the mesh....done.  [N=117,L=206]  [time=0.099601]
MAIN: refining the mesh....done.  [N=219,L=406]  [time=5.64105]
MAIN: plotting the manifold....done.  [N=219,L=406]
 --- press return ---
..done.  [N=219,L=406]
 --- press return ---
MAIN: plotting the mesh....done.  [N=219,L=406]
 --- press return ---
MAIN: nonlinear solve starting.  [N=219,L=406]
NEWTON: iter = 0, ||F(u)|| = 1.91734
NEWTON: iter = 1, ||F(u)|| = 2.9574e-15
MAIN: nonlinear solve finished.  [time=3.25056]
MAIN: discretization error is = 0.106009
MAIN: graphing matrix....done.  [N=219,L=406]
 --- press return ---
MAIN: showing nonzero structure....done.  [N=219,L=406]
 --- press return ---
MAIN: plotting FEM solution....done.  [N=219,L=406]
 --- press return ---
```

Acknowledgments

The author thanks R. Bank for many enlightening discussions.

References

1. R. A. Adams, *Sobolev spaces*, Academic Press, San Diego, CA, 1978.
2. D.N. Arnold, A. Mukherjee, and L. Pouly, *Locally adapted tetrahedral meshes using bisection*, 1997.
3. I. Babuška and W.C. Rheinboldt, *Error estimates for adaptive finite element computations*, SIAM J. Numer. Anal. **15** (1978), 736–754.
4. ———, *A posteriori error estimates for the finite element method*, Int. J. Numer. Meth. Engrg. **12** (1978), 1597–1615.
5. R. Bank and M. Holst, *A new paradigm for parallel adaptive mesh refinement*, SIAM J. Sci. Statist. Comput. (Accepted; to appear).
6. R. E. Bank, *PLTMG: A software package for solving elliptic partial differential equations, users' guide 8.0*, Software, Environments and Tools, Vol. 5, SIAM, Philadelphia, PA, 1998.
7. R. E. Bank and H. D. Mittelmann, *Stepsize selection in continuation procedures and damped Newton's method*, J. Computational and Applied Mathematics **26** (1989), 67–77.
8. R. E. Bank and D. J. Rose, *Parameter selection for Newton-like methods applicable to nonlinear partial differential equations*, SIAM J. Numer. Anal. **17** (1980), no. 6, 806–822.
9. ———, *Global Approximate Newton Methods*, Numer. Math. **37** (1981), 279–295.
10. ———, *Analysis of a multilevel iterative method for nonlinear finite element equations*, Math. Comp. **39** (1982), no. 160, 453–465.
11. R. E. Bank and R. K. Smith, *A posteriori error estimates based on hierarchical bases*, SIAM J. Numer. Anal. **30** (1993), no. 4, 921–935.
12. ———, *Mesh smoothing using* a posteriori *error estimates*, SIAM J. Numer. Anal. **34** (1997), 979–997.
13. R. E. Bank and A. Weiser, *Some a posteriori error estimators for elliptic partial differential equations*, Math. Comp. **44** (1985), no. 170, 283–301.

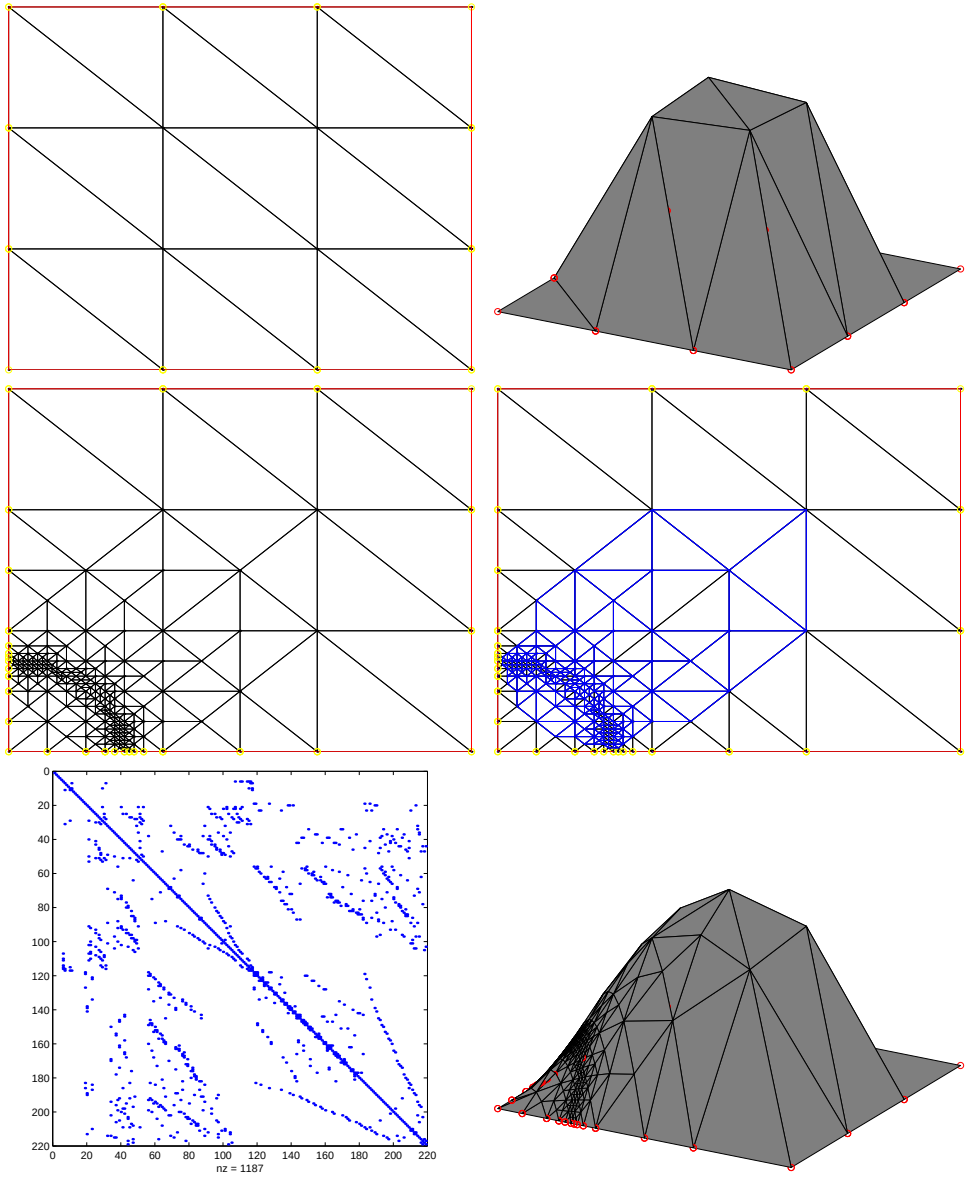

FIGURE 4. MCLite graphical output.

14. E. Bänsch, *An adaptive finite-element strategy for the three-dimsional time-dependent Navier-Stokes equations*, Journal of Computional and Applied Mathematics **36** (1991), 3–28.
15. ———, *Local mesh refinement in 2 and 3 dimensions*, Impact of Computing in Science and Engineering **3** (1991), 181–191.
16. P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners, *UG – a flexible software toolbox for solving partial differential equations*, 1998.
17. R. Beck, B. Erdmann, and R. Roitzsch, *KASKADE 3.0: An ojbect-oriented adaptive finite element code*, Tech. Report TR95–4, Konrad-Zuse-Zentrum for Informationstechnik, Berlin, 1995.
18. J. Bey, *Adaptive grid manager: AGM3D manual*, Tech. Report 50, SFB 382, Math. Inst. Univ. Tubingen, 1996.
19. S. C. Brenner and L. R. Scott, *The mathematical theory of finite element methods*, Springer-Verlag, New York, NY, 1994.
20. P. G. Ciarlet, *The finite element method for elliptic problems*, North-Holland, New York, NY, 1978.
21. P. J. Davis, *Interpolation and approximation*, Dover Publications, Inc., New York, NY, 1963.
22. R. S. Dembo, S. C. Eisenstat, and T. Steihaug, *Inexact Newton Methods*, SIAM J. Numer. Anal. **19** (1982), no. 2, 400–408.
23. R. A. DeVore and G. G. Lorentz, *Constructive approximation*, Springer-Verlag, New York, NY, 1993.
24. S. C. Eisenstat and H. F. Walker, *Globally Convergent Inexact Newton Methods*, Tech. report, Dept. of Mathematics and Statistics, Utah State University, 1992.
25. S. Fucik and A. Kufner, *Nonlinear differential equations*, Elsevier Scientific Publishing Company, New York, NY, 1980.

26. W. Hackbusch, *Multi-grid methods and applications*, Springer-Verlag, Berlin, Germany, 1985.
27. ———, *Elliptic differential equations*, Springer-Verlag, Berlin, Germany, 1992.
28. ———, *Iterative solution of large sparse systems of equations*, Springer-Verlag, Berlin, Germany, 1994.
29. M. Holst, *Adaptive multilevel finite element methods on manifolds and their implementation in MC*, (In preparation; currently available as a technical report and User's Guide to the MC software).
30. H. B. Keller, *Numerical methods in bifurcation problems*, Tata Institute of Fundamental Research, Bombay, India, 1987.
31. A. Liu and B. Joe, *Relationship between tetrahedron shape measures*, BIT **34** (1994), 268–287.
32. ———, *Quality local refinement of tetrahedral meshes based on bisection*, SIAM J. Sci. Statist. Comput. **16** (1995), no. 6, 1269–1291.
33. J.L. Liu and W.C. Rheinboldt, *A posteriori finite element error estimators for indefinite elliptic boundary value problems*, Numer. Funct. Anal. and Optimiz. **15** (1994), no. 3, 335–356.
34. ———, *A posteriori finite element error estimators for parametrized nonlinear boundary value problems*, Numer. Funct. Anal. and Optimiz. **17** (1996), no. 5, 605–637.
35. J.M. Maubach, *Local bisection refinement for N-simplicial grids generated by relection*, SIAM J. Sci. Statist. Comput. **16** (1995), no. 1, 210–277.
36. E.P. Mucke, *Shapes and implementations in three-dimensional geometry*, Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1993.
37. A. Mukherjee, *An adaptive finite element code for elliptic boundary value problems in three dimensions with applications in numerical relativity*, Ph.D. thesis, Dept. of Mathematics, The Pennsylvania State University, 1996.
38. M.C. Rivara, *Algorithms for refining triangular grids suitable for adaptive and multigrid techniques*, International Journal for Numerical Methods in Engineering **20** (1984), 745–756.
39. ———, *Local modification of meshes for adaptive and/or multigrid finite-element methods*, Journal of Computational and Applied Mathematics **36** (1991), 79–89.
40. I.G. Rosenberg and F. Stenger, *A lower bound on the angles of triangles constructed by bisecting the longest side*, Math. Comp. **29** (1975), 390–395.
41. M. Stynes, *On faster convergence of the bisection method for all triangles*, Math. Comp. **35** (1980), 1195–1201.
42. R. S. Varga, *Matrix iterative analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1962.
43. R. Verfürth, *A posteriori error estimates for nonlinear problems. Finite element discretizations of elliptic equations*, Math. Comp. **62** (1994), no. 206, 445–475.
44. ———, *A review of* a posteriori *error estimation and adaptive mesh-refinement techniques*, John Wiley & Sons Ltd, New York, NY, 1996.
45. J. Xu and A. Zhou, *Local error estimates and parallel adaptive algorithms*, 1997, Preprint.

Department of Mathematics, University of California at San Diego, La Jolla, CA 92093, USA.
E-mail address: mholst@math.ucsd.edu