Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Charm++ Iterative Finite Element Matrix (IFEM) Library Manual

Initial version of Charm++ Finite Element Framework was developed by Orion Lawlor in the spring of 2003.

Version 1.2

University of Illinois
Charm++/Converse Parallel Programming System Software
Non-Exclusive, Non-Commercial Use License

Upon execution of this Agreement by the party identified below ("Licensee"), The Board of Trustees of the University of Illinois ("Illinois"), on behalf of The Parallel Programming Laboratory ("PPL") in the Department of Computer Science, will provide the Charm++/Converse Parallel Programming System software ("Charm++") in Binary Code and/or Source Code form ("Software") to Licensee, subject to the following terms and conditions. For purposes of this Agreement, Binary Code is the compiled code, which is ready to run on Licensee's computer. Source code consists of a set of files which contain the actual program commands that are compiled to form the Binary Code.

1. The Software is intellectual property owned by Illinois, and all right, title and interest, including copyright, remain with Illinois. Illinois grants, and Licensee hereby accepts, a restricted, non-exclusive, non-transferable license to use the Software for academic, research and internal business purposes only, e.g. not for commercial use (see Clause 7 below), without a fee.

2. Licensee may, at its own expense, create and freely distribute complimentary works that interoperate with the Software, directing others to the PPL server (`http://charm.cs.uiuc.edu`) to license and obtain the Software itself. Licensee may, at its own expense, modify the Software to make derivative works. Except as explicitly provided below, this License shall apply to any derivative work as it does to the original Software distributed by Illinois. Any derivative work should be clearly marked and renamed to notify users that it is a modified version and not the original Software distributed by Illinois. Licensee agrees to reproduce the copyright notice and other proprietary markings on any derivative work and to include in the documentation of such work the acknowledgement:

   > "This software includes code developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Licensee may redistribute without restriction works with up to 1/2 of their non-comment source code derived from at most 1/10 of the non-comment source code developed by Illinois and contained in the Software, provided that the above directions for notice and acknowledgement are observed. Any other distribution of the Software or any derivative work requires a separate license with Illinois. Licensee may contact Illinois (`kale@cs.uiuc.edu`) to negotiate an appropriate license for such distribution.

3. Except as expressly set forth in this Agreement, THIS SOFTWARE IS PROVIDED "AS IS" AND ILLINOIS MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS. LICENSEE AGREES THAT UNIVERSITY SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY LICENSEE OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS AGREEMENT OR USE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS.

4. Licensee understands the Software is proprietary to Illinois. Licensee agrees to take all reasonable steps to insure that the Software is protected and secured from unauthorized disclosure, use, or release and will treat it with at least the same level of care as Licensee would use to protect and secure its own proprietary computer programs and/or information, but using no less than a reasonable standard of care. Licensee agrees to provide the Software only to any other person or entity who has registered with Illinois. If licensee is not registering as an individual but as an institution or corporation each member of the institution or corporation who has access to or uses Software must agree to and abide by the terms of this license. If Licensee becomes aware of any unauthorized licensing, copying or use of the Software, Licensee shall promptly notify Illinois in writing. Licensee expressly agrees to use the Software only in the manner and for the specific uses authorized in this Agreement.

5. By using or copying this Software, Licensee agrees to abide by the copyright law and all other applicable laws of the U.S. including, but not limited to, export control laws and the terms of this license. Illinois shall have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with, any terms of the license. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by its failure to abide by the terms of this license. Upon termination, Licensee agrees to destroy all copies of the Software in its possession and to verify such destruction in writing.

6. The user agrees that any reports or published results obtained with the Software will acknowledge its use by the appropriate citation as follows:

   > "Charm++/Converse was developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Any published work which utilizes Charm++ shall include the following reference:

   > "L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In 'Parallel Programming using C++' (Eds. Gregory V. Wilson and Paul Lu), pp 175-213, MIT Press, 1996."

   Any published work which utilizes Converse shall include the following reference:

   > "L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. Proceedings of the 10th International Parallel Processing Symposium, pp 212-217, April 1996."

   Electronic documents will include a direct link to the official Charm++ page at `http://charm.cs.uiuc.edu/`

7. Commercial use of the Software, or derivative works based thereon, REQUIRES A COMMERCIAL LICENSE. Should Licensee wish to make commercial use of the Software, Licensee will contact Illinois (kale@cs.uiuc.edu) to negotiate an appropriate license for such use. Commercial use includes:

   (a) integration of all or part of the Software into a product for sale, lease or license by or on behalf of Licensee to third parties, or

   (b) distribution of the Software to third parties that need it to commercialize product sold or licensed by or on behalf of Licensee.

8. Government Rights. Because substantial governmental funds have been used in the development of Charm++/Converse, any possession, use or sublicense of the Software by or to the United States government shall be subject to such required restrictions.

9. Charm++/Converse is being distributed as a research and teaching tool and as such, PPL encourages contributions from users of the code that might, at Illinois' sole discretion, be used or incorporated to make the basic operating framework of the Software a more stable, flexible, and/or useful product. Licensees who contribute their code to become an internal portion of the Software agree that such code may be distributed by Illinois under the terms of this License and may be required to sign an "Agreement Regarding Contributory Code for Charm++/Converse Software" before Illinois can accept it (contact `kale@cs.uiuc.edu` for a copy).

UNDERSTOOD AND AGREED.
Contact Information:
The best contact path for licensing issues is by e-mail to `kale@cs.uiuc.edu` or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 333-3501

Contents

1 Introduction

This manual presents the Iterative Finite Element Matrix (IFEM) library, a library for easily solving matrix problems derived from finite-element formulations. The library is designed to be matrix-free, in that the only matrix operation required is matrix-vector product, and hence the entire matrix need never be assembled

IFEM is built on the mesh and communication capabilities of the Charm++ FEM Framework, so for details on the basic runtime, problem setup, and partitioning see the FEM Framework manual.

1.1 Terminology

A FEM program manipulates elements and nodes. An **element** is a portion of the problem domain, also known as a cell, and is typically some simple shape like a triangle, square, or hexagon in 2D; or tetrahedron or rectangular solid in 3D. A **node** is a point in the domain, and is often the vertex of several elements. Together, the elements and nodes form a **mesh**, which is the central data structure in the FEM framework. See the FEM manual for details.

2 Solvers

A IFEM **solver** is a subroutine that controls the search for the solution.

Solvers often take extra parameters, which are listed in a type called in C ILSI_Param, which in Fortran is an array of ILSI_PARAM doubles. You initialize these solver parameters using the subroutine ILSI_Param_new, which takes the parameters as its only argument. The input and output parameters in an ILSI_Param are listed in Table 1 and Table 2.

C Field Name	Fortran Field Offset	Use
maxResidual	1	If nonzero, termination criteria: magnitude of residual.
maxIterations	2	If nonzero, termination criteria: number of iterations.
solverIn[8]	3-10	Solver-specific input parameters.

Table 1: ILSI_Param solver input parameters.

C Field Name	Fortran Field Offset	Use
residual	11	Magnitude of residual of final solution.
iterations	12	Number of iterations actually taken.
solverOut[8]	13-20	Solver-specific output parameters.

Table 2: ILSI_Param solver output parameters.

2.1 Conjugate Gradient Solver

The only solver currently written using IFEM is the conjugate gradient solver. This linear solver requires the matrix to be real, symmetric and positive definite.

Each iteration of the conjugate gradient solver requires one matrix-vector product and two global dot products. For well-conditioned problems, the solver typically converges in some small multiple of the diameter of the mesh–the number of elements along the largest side of the mesh.

You access the conjugate gradient solver via the subroutine name ILSI_CG_Solver.

3 Solving Shared-Node Systems

Many problems encountered in FEM analysis place the entries of the known and unknown vectors at the nodes–the vertices of the domain. Elements provide linear relationships between the known and unknown node values, and the entire matrix expresses the combination of all these element relations.

For example, in a structural statics problem, we know the net force at each node, f, and seek the displacements of each node, u. Elements provide the relationship, often called a stiffness matrix K, between a nodes' displacments and its net forces:

$$f = Ku$$

We normally label the known vector b (in the example, the forces), the unknown vector x (in the example, the displacments), and the matrix A:

$$b = Ax$$

IFEM provides two routines for solving problems of this type. The first routine, IFEM_Solve_shared, solves for the entire x vector based on the known values of the b vector. The second, IFEM_Solve_shared_bc, allows certain entries in the x vector to be given specific values before the problem is solved, creating values for the b vector.

3.1 IFEM_Solve_shared

void IFEM_Solve_shared(ILSI_Solver s,ILSI_Param *p, int fem_mesh, int fem_entity,int length,int width, IFEM_Matrix_product_c A, void *ptr, const double *b, double *x);
subroutine IFEM_Solve_shared(s,p, fem_mesh,fem_entity,length,width, A,ptr,b,x)

```
    external solver subroutine ::  s
    double precision, intent(inout) ::  p(ILSI_PARAM)
    integer, intent(in) ::  fem_mesh, fem_entity, length,width
    external matrix-vector product subroutine ::  A
    TYPE(varies), pointer ::  ptr
    double precision, intent(in) ::  b(width,length)
    double precision, intent(inout) ::  x(width,length)
```

This routine solves the linear system $Ax = b$ for the unknown vector x. s and p give the particular linear solver to use, and are described in more detail in Section 2. *fem_mesh* and *fem_entity* give the FEM framework mesh (often FEM_Mesh_default_read()) and entity (often FEM_NODE) with which the known and unknown vectors are listed.

width gives the number of degrees of freedom (entries in the vector) per node. For example, if there is one degree of freedom per node, width is one. *length* should always equal the number of FEM nodes.

A is a local matrix-vector product routine you must write. Its interface is described in Section 3.1.1. *ptr* is a pointer passed down to A–it is not otherwise used by the framework.

b is the known vector. x, on input, is the initial guess for the unknown vector. On output, x is the final value for the unknown vector. b and x should both have length * width entries. In C, DOF i of node n should be indexed as $x[n*width+i]$. In Fortran, these arrays should be allocated like *x(width,length)*.

When this routine returns, x is the final value for the unknown vector, and the output values of the solver parameters p will have been written.

```
// C++ Example
  int mesh=FEM_Mesh_default_read();
  int nNodes=FEM_Mesh_get_length(mesh,FEM_NODE);
  int width=3; //A 3D problem
  ILSI_Param solverParam;
  struct myProblemData myData;

  double *b=new double[nNodes*width];
  double *x=new double[nNodes*width];
  ... prepare solution target b and guess x ...

  ILSI_Param_new(&solverParam);
  solverParam.maxResidual=1.0e-4;
  solverParam.maxIterations=500;

  IFEM_Solve_shared(IFEM_CG_Solver,&solverParam,
         mesh,FEM_NODE, nNodes,width,
         myMatrixVectorProduct, &myData, b,x);

! F90 Example
  include 'ifemf.h'
  INTEGER :: mesh, nNodes,width
  DOUBLE PRECISION, ALLOCATABLE :: b(:,:), x(:,:)
  DOUBLE PRECISION :: solverParam(ILSI_PARAM)
  TYPE(myProblemData) :: myData
```

```
mesh=FEM_Mesh_default_read()
nNodes=FEM_Mesh_get_length(mesh,FEM_NODE)
width=3   ! A 3D problem

ALLOCATE(b(width,nNodes), x(width,nNodes))
... prepare solution target b and guess x ..

ILSI_Param_new(&solverParam);
solverParam(1)=1.0e-4;
solverParam(2)=500;

IFEM_Solve_shared(IFEM_CG_Solver,solverParam,
       mesh,FEM_NODE, nNodes,width,
       myMatrixVectorProduct, myData, b,x);
```

3.1.1 Matrix-vector product routine

IFEM requires you to write a matrix-vector product routine that will evaluate Ax for various vectors x. You may use any subroutine name, but it must take these arguments:

```
void IFEM_Matrix_product(void *ptr,int length,int width, const double *src, double *dest);
subroutine IFEM_Matrix_product(ptr,length,width,src,dest)
    TYPE(varies), pointer ::  ptr
    integer, intent(in) ::  length,width
    double precision, intent(in) ::  src(width,length)
    double precision, intent(out) ::  dest(width,length)
```

The framework calls this user-written routine when it requires a matrix-vector product. This routine should compute $dest = A\,src$, interpreting *src* and *dest* as vectors. *length* gives the number of nodes and *width* gives the number of degrees of freedom per node, as above.

In writing this routine, you are responsible for choosing a representation for the matrix A. For many problems, there is no need to represent A explicitly–instead, you simply evaluate *dest* by looping over local elements, taking into account the values of *src*. This example shows how to write the matrix-vector product routine for simple 1D linear elastic springs, while solving for displacement given net forces.

After calling this routine, the framework will handle combining the overlapping portions of these vectors across processors to arrive at a consistent global matrix-vector product.

```
// C++ Example
#include "ifemc.h"

typedef struct {
  int nElements; //Number of local elements
  int *conn; // Nodes adjacent to each element: 2*nElements entries
  double k; //Uniform spring constant
} myProblemData;

void myMatrixVectorProduct(void *ptr,int nNodes,int dofPerNode,
          const double *src,double *dest)
{
  myProblemData *d=(myProblemData *)ptr;
  int n,e;
  // Zero out output force vector:
  for (n=0;n<nNodes;n++) dest[n]=0;
  // Add in forces from local elements
  for (e=0;e<d->nElements;e++)
    int n1=d->conn[2*e+0]; // Left node
    int n2=d->conn[2*e+1]; // Right node
    double f=d->k * (src[n2]-src[n1]); //Force
    dest[n1]+=f;
    dest[n2]-=f;

}


! F90 Example
  TYPE(myProblemData)
    INTEGER :: nElements
    INTEGER, ALLOCATABLE :: conn(2,:)
    DOUBLE PRECISION :: k
```

```
  END TYPE

SUBROUTINE myMatrixVectorProduct(d,nNodes,dofPerNode,src,dest)
  include 'ifemf.h'
  TYPE(myProblemData), pointer :: d
  INTEGER :: nNodes,dofPerNode
  DOUBLE PRECISION :: src(dofPerNode,nNodes), dest(dofPerNode,nNodes)
  INTEGER :: e,n1,n2
  DOUBLE PRECISION :: f

  dest(:,:)=0.0
  do e=1,d%nElements
    n1=d%conn(1,e)
    n2=d%conn(2,e)
    f=d%k * (src(1,n2)-src(1,n1))
    dest(1,n1)=dest(1,n1)+f
    dest(1,n2)=dest(1,n2)+f
  end do
END SUBROUTINE
```

3.2 IFEM_Solve_shared_bc

void IFEM_Solve_shared_bc(ILSI_Solver s,ILSI_Param *p, int fem_mesh, int fem_entity,int length,int width, int bcCount, const int *bcDOF, const double *bcValue, IFEM_Matrix_product_c A, void *ptr, const double *b, double *x);
subroutine IFEM_Solve_shared_bc(s,p, fem_mesh,fem_entity,length,width, bcCount,bcDOF,bcValue, A,ptr,b,x)

```
    external solver subroutine ::  s
    double precision, intent(inout) ::  p(ILSI_PARAM)
    integer, intent(in) ::  fem_mesh, fem_entity, length,width
    integer, intent(in) ::  bcCount
    integer, intent(in) ::  bcDOF(bcCount)
    double precision, intent(in) ::  bcValue(bcCount)
    external matrix-vector product subroutine ::  A
    TYPE(varies), pointer ::  ptr
    double precision, intent(in) ::  b(width,length)
    double precision, intent(inout) ::  x(width,length)
```

Like IFEM_Solve_shared, this routine solves the linear system $Ax = b$ for the unknown vector x. This routine, however, adds support for boundary conditions associated with x. These so-called "essential" boundary conditions restrict the values of some unknowns. For example, in structural dynamics, a fixed displacment is such an essential boundary condition.

The only form of boundary condition currently supported is to impose a fixed value on certain unknowns, listed by their degree of freedom–that is, their entry in the unknown vector. In general, the i'th DOF of node n has DOF number $n * width + i$ in C and $(n-1) * width + i$ in Fortran. The framework guarantees that, on output, for all $bcCount$ boundary conditions, $x(bcDOF(f)) = bcValue(f)$.

For example, if $width$ is 3 in a 3d problem, we would set node ny's y coordinate to 4.6 and node nz's z coordinate to 7.3 like this:

```
// C++ Example
  int bcCount=2;
  int bcDOF[bcCount];
  double bcValue[bcCount];
  // Fix node ny's y coordinate
  bcDOF[0]=ny*width+1; // y is coordinate 1
  bcValue[0]=4.6;
  // Fix node nz's z coordinate
  bcDOF[1]=nz*width+2; // z is coordinate 2
  bcValue[1]=2.0;

! F90 Example
// C++ Example
  integer :: bcCount=2;
  integer :: bcDOF(bcCount);
  double precision :: bcValue(bcCount);
  // Fix node ny's y coordinate
  bcDOF(1)=(ny-1)*width+2; // y is coordinate 2
  bcValue(1)=4.6;
  // Fix node nz's z coordinate
  bcDOF(2)=(nz-1)*width+3; // z is coordinate 3
  bcValue(2)=2.0;
```

Mathematically, what is happening is we are splitting the partially unknown vector x into a completely unknown portion y and a known part f:

$$Ax = b$$

$$A(y + f) = b$$
$$Ay = b - Af$$

We can then define a new right hand side vector $c = b - Af$ and solve the new linear system $Ay = c$ normally. Rather than renumbering, we do this by zeroing out the known portion of x to make y. The creation of the new linear system, and the substitution back to solve the original system are all done inside this subroutine.

One important missing feature is the ability to specify general linear constraints on the unknowns, rather than imposing specific values.

4 Index