

SolverBlaze (tm) Finite Element Library

Computations & Graphics, Inc. (CGI) SolverBlaze Finite Element Library is a powerful structural and finite element analysis API (Application Programming Interface). It is based on the time-tested finite element solver engine in Real3D-Analysis, which has been used by hundreds of civil and structural engineers. You can use the API to develop your custom software royalty-free.

The following are the main features:

- Supports beam, truss, plate and shell (thin and thick, compatible and incompatible formulations) and brick elements.
- Supports linear and nonlinear nodal, linear and surface springs.
- Supports nodal, point, line, surface and area loads.
- Supports static linear and P-Delta analysis and frequency analysis.
- Support both skyline and extremely fast sparse solver
- Bidirectional support to Real3D-Analysis so you can graphically view your generated model and results.
- Supports both native C++ language and managed .NET languages such as C# and VB.NET.
- Supports Unicode and Non-Unicode in Visual Studio 2005, 2008 and 2010.
- Supports Linux and Mac OS (source code only).
- Available in both binary library and source code form.
- A free copy of Professional Real3D-Analysis Program.
- Reasonably priced.

C++ Interface

The C++ library is a 32-bit Windows DLL (cgiSolverBlaze.dll). The interface includes the following header files: `_cgiDefines.h`, `_cgiIStructure.h`, `_cgiPlatform.h` and `cgiSolverBlaze.h`. It also includes a `cgiSolverBlaze.lib` for linking to your projects. Intel Parsido sparse solver DLL (`libguide40.dll`) is also included.

The `_cgiDefines.h` defines all input and output data structures. `_cgiIStructure` is the one and only interface to set input, perform analysis and retrieve output. The best way to learn how you use these data structures and interfaces is to study the examples included in the C++ console application project `cgiSolverBlazeTest`. These examples are taken from the Verification Manual of Real3D-Analysis, which is a structural analysis and finite element analysis program by CGI. These examples include 2D/3D frame analysis, plate bending analysis and frequency analysis. A full input and output report is produced after each analysis. You can compare the reports with those produced from within Real3D-Analysis.

Four versions of `cgiSolverBlaze.lib` and `cgiSolverBlaze.dll` are provided for your configuration needs: Debug Unicode, Release Unicode, Debug Non-Unicode and Release Non-Unicode. You should link your projects to the correct version of the lib file. Make sure you also copy the correct version of the DLLs (`libguide40.dll`, `cgiSolverBlaze.dll`) to your executable directory.

To start, you create a `cgiIStructure` interface object like this:

```
cgiIStructure* pStructure = CreateStructure();
```

You can supply three optional callback functions for the solver to notify your application:

```
typedef void (* fnLISTMSG)(LPCTSTR sz0, LPCTSTR sz1);
typedef void (* fnSTATUSMSG)(LPCTSTR sz);
typedef int (*fnMKLPROGRESS)( int* ithr, int* step, char* stage, int len );

pStructure->setListMessageFunction(ListMsg);
pStructure->setStatusMessageFunction(StatusMsg);
pStructure->setSparseSolverProgressFunction(MkProgress);
```

Next you can set the model type as defined in the `_cgiDefines.h`. For example:

```
pStructure->setModelType(kModel_Frame2D); // a 2D Frame
```

Four unit systems are available in the library. They are Standard English unit, Standard Metric unit, Consistent English unit and Consistent Metric unit. For Standard English unit system, the following units are used in length, section dimension, force etc.

```
LENGTH=ft; DIMENSION=in;
FORCE=kip; FORCE_LINE=kip/ft; MOMENT=kip-ft; FORCE_SURFACE=lb/ft^2;
DISPLACEMENT_TRANS=in; DISPLACEMENT_ROTATE=rad;
MODULUS=kip/in^2; WEIGHT_DENSITY=lb/ft^3; STRESS=lb/in^2;
SPRING_TRANS_1D=lb/in; SPRING_ROTATE_1D=lb-in/rad; SPRING_TRANS_2D=kip/in^2;
SPRING_TRANS_3D=kip/in^3
pStructure->setStandardEnglishUnits();
```

The input includes definitions for materials, sections, nodes, elements, load cases, load combinations, loads (nodal, point, line etc) in each load case. You can also set different analysis and report options. For example:

```

// define materials
vector<cgiMaterial> vMat;
cgiMaterial mat;
mat.setId(1); // material id, to be referred later
// material label, young's modulus, poisson ratio, weight density
mat.setProperties(_T("Default"), 29000, 0.3, 450);
vMat.push_back(mat);
pStructure->setMaterials(vMat);

// define sections
vector<cgiSection> vSect;
cgiSection sect;
sect.setId(1);
vSect.push_back(sect); // default section, we are not going to use it
sect.setId(2); // section id
_tcscpy(sect.szLabel, _T("W27X84")); // section label, can not contain spaces
sect.fVal[cgiSection::A] = 24.8; // sectional area: in^2
sect.fVal[cgiSection::Ayy] = 12.282; // major shear area: in^2
sect.fVal[cgiSection::Azz] = 12.7488; // minor shear area: in^2
sect.fVal[cgiSection::Izz] = 2850; // major moment of inertia: in^4
sect.fVal[cgiSection::Iyy] = 106; // minor moment of inertia: in^4
sect.fVal[cgiSection::J] = 2.81; // rotational moment of inertia: in^4
// or you can properties like this
//sect.setProperties(_T("W27X84"), 24.8, 12.282, 12.7488, 2850, 106, 2.81);
vSect.push_back(sect);
sect.setId(3); // section id
sect.setProperties(_T("W10X45"), 13.3, 3.535, 9.9448, 248, 53.4, 1.51);
vSect.push_back(sect);
pStructure->setSections(vSect);

// define nodes
vector<cgiNode> vNd;
cgiNode nd;
nd.setId(1); // nodal id
nd.setCoordinates(0, 0, 0); // nodal coordinates
vNd.push_back(nd);
nd.setId(2);
nd.setCoordinates(0, 12, 0);
vNd.push_back(nd);
nd.setId(3);
nd.setCoordinates(0, 24, 0);
vNd.push_back(nd);

nd.setId(5);
nd.setCoordinates(60, 0, 0);
vNd.push_back(nd);
nd.setId(6);
nd.setCoordinates(60, 12, 0);
vNd.push_back(nd);

nd.setId(4);
nd.setCoordinates(60, 24, 0);
vNd.push_back(nd);
pStructure->setNodes(vNd);

// define beams
vector<cgiBeam> vBm;
cgiBeam bm;
bm.setId(1); // beam id
bm.setNodes(1, 2); // begin and end node ids of the beam
bm.setProperties(1, 3, 0); // material id, section id, beam angle in radian
vBm.push_back(bm);
bm.setId(2);
bm.setNodes(2, 3);
bm.setProperties(1, 3, 0);
vBm.push_back(bm);
bm.setId(3);
bm.setNodes(3, 4);
bm.setProperties(1, 2, 0);
vBm.push_back(bm);

bm.setId(5);
bm.setNodes(6, 4);
bm.setProperties(1, 3, 0);
vBm.push_back(bm);

bm.setId(4);
bm.setNodes(5, 6);

```

```

bm.setProperties(1, 3, 0);
vBm.push_back(bm);
pStructure->setBeams(vBm);

// define supports
vector<cgiSupport> vSupt;
cgiSupport supt;
supt.setId(1); // nodal id that this support is on
// six DOFs, 1 for supported, 0 for free; forced settlements and rotations

supt.setSupportDOFs(_T("111000"), 0, 0, 0, 0, 0, 0);
vSupt.push_back(supt);
supt.setId(5);
supt.setSupportDOFs(_T("111000")); // forced settlements and rotations default to 0s
vSupt.push_back(supt);
pStructure->setSupports(vSupt);

// define load cases
vector<cgiLoadCase> vLoadCase;
cgiLoadCase loadcase;
loadcase.setId(1); // load case id
// load case label; type such as DEAD, LIVE etc; report on this load case
loadcase.setLoadCase(_T("Default"), _T("DEAD"), TRUE);
vLoadCase.push_back(loadcase);
pStructure->setLoadCases(vLoadCase);

// define load combinations
vector<cgiLoadComb> vLoadComb;
cgiLoadComb loadcomb; // load combination
loadcomb.clearLoadCombItem(); // make sure we start clean with this load combination
// load combination label, linear combination (no pdelta effect), report
loadcomb.setLoadComb(_T("Linear"), FALSE, TRUE);
// add load case and factor to this load combination
loadcomb.addLoadCombItem(_T("Default"), 1.0);
vLoadComb.push_back(loadcomb);

loadcomb.clearLoadCombItem(); // make sure we start clean with this load combination
loadcomb.setLoadComb(_T("P-Delta"), TRUE, TRUE);
loadcomb.addLoadCombItem(_T("Default"), 1.0);
vLoadComb.push_back(loadcomb);
pStructure->setLoadCombinations(vLoadComb);

// each case load corresponds to each load case
// each case load includes nodal loads, point loads, line loads etc in this load case

vector<cgiCaseLoad> vCaseLoad;
cgiCaseLoad caseload;
cgiNodalLoad ndload;
ndload.setLoad(4, -120.0, Y);
caseload.addNodalLoad(ndload);
ndload.setLoad(3, 6.0, X); // node id, load magnitude and direction
caseload.addNodalLoad(ndload);

cgiPointLoad ptload;
// member id, load magnitude, distance from member start in percentage/100, load
direction
ptload.setLoad(3, GLOBAL, -60.0, 0.333333, Y);
caseload.addPointLoad(ptload);
vCaseLoad.push_back(caseload);
pStructure->setCaseLoads(vCaseLoad);

// set report options
cgiReportOptions reportOptions;
reportOptions.SelectAll();
reportOptions.bHTML = FALSE;
pStructure->setReportOptions(reportOptions);

// set analysis options
bool bConsiderBeamShearDeformation = FALSE;
int nMaximumPDeltaIterations = 10;
double fPDeltaToleranceInPercentage = 0.5; // in percentage
int nNumberOfSegmentsForBeamOutput = 20;
bool bUseThinPlate = false;
bool bUseCompatibleModes = true;
int nUseAverageStressMode = 1;
pStructure->setAnalysisOptions(bConsiderBeamShearDeformation, nMaximumPDeltaIterations,
                             fPDeltaToleranceInPercentage,
                             nNumberOfSegmentsForBeamOutput, bUseThinPlate,

```

```

        bUseCompatibleModes, nUseAverageStressMode);
pStructure->setSolver(kSolverSparse64);

```

Before you perform structural analysis, you can save the input to a file, which can then be opened by Real3D-Analysis. This can be very helpful in visualizing your input and its results. You can then run report command which will produce a HTML report file.

```

// save the data to a file for possible later use
bool b2 = pStructure->saveDocument(_T("c:\\temp\\Verify-Example4.r3a"));

// run static analysis
bool bRun = pStructure->runStaticAnalysis();
if(!bRun)
{
    COUT << _T("Error running static analysis... ") << _T("c:\\temp\\Verify-
        Example4.r3a") << endl;
    return;
}

// report
pStructure->setListMessageFunction(0); // do not list message
if(!pStructure->runReport())
{
    COUT << _T("Error running report... ") << _T("c:\\temp\\Verify-Example4.r3a") <<
        endl;
    return;
}

```

The analysis results can be retrieved easily as illustrated in the following.

```

// extract results
const std::vector<cgiCombResult>& vCombResult = pStructure->getStaticResults();
// list displacements for all load combinations
TCHAR szTranslationalDisplacementUnit[LABEL_SIZE];
TCHAR szRotationalDisplacementUnit[LABEL_SIZE];
pStructure->getUnit(szTranslationalDisplacementUnit, DISPLACEMENT_TRANS);
pStructure->getUnit(szRotationalDisplacementUnit, DISPLACEMENT_ROTATE);
COUT << _T("-----") << endl;
COUT << _T("----- Verifying Example 4 -----") << endl;
COUT << endl << _T("Nodal Displacements. Units: ") << szTranslationalDisplacementUnit
    << _T(", ") << szRotationalDisplacementUnit << endl;
for(int iComb = 0; iComb < vCombResult.size(); iComb++)
{
    COUT << _T("Load combination ") << iComb + 1 << ": " << vLoadComb[iComb].szLabel
        << endl;

    int nDisps = vCombResult[iComb].m_vNdDisp.size();
    COUT << setprecision(4) << fixed;
    for(int i = 0; i < nDisps; i++) {
        const cgiResult6Val& disp = vCombResult[iComb].m_vNdDisp[i];
        COUT << _T("Node- ") << disp.iId << _T(": ")
            << disp.fVal[X] << "\\t" << disp.fVal[Y] << "\\t" << disp.fVal[Z] << "\\t"
            << disp.fVal[OX] << "\\t" << disp.fVal[OY] << "\\t" << disp.fVal[OZ] << endl;
    }
    COUT << endl;
}

// list internal forces and moments at beam ends for all load combinations
TCHAR szForceUnit[LABEL_SIZE];
TCHAR szMomentUnit[LABEL_SIZE];
pStructure->getUnit(szForceUnit, FORCE);
pStructure->getUnit(szMomentUnit, MOMENT);
COUT << endl << _T("Beam End Forces and Moments. Units: ") << szForceUnit << _T(", ")
    << szMomentUnit << endl;
for(int iComb = 0; iComb < vCombResult.size(); iComb++)
{
    COUT << _T("Load combination ") << iComb + 1 << ": " << vLoadComb[iComb].szLabel
        << endl;

    int nBeams = vCombResult[iComb].m_vBnVMD.size();

```

```

for(int i = 0; i < nBeams; i++) {
    const cgiBeamVMD& bmVMD = vCombResult[iComb].m_vBmVMD[i];

    int iSeg[2];
    iSeg[0] = 0;
    iSeg[1] = (int)bmVMD.vVMD.size() - 1;
    COUT << setprecision(4) << fixed;
    for(int k = 0; k < 2; k++)
    {
        const cgiVMD& vmd = bmVMD.vVMD[iSeg[k]];
        TCHAR* szEnd = (k==0)? _T("Start") : _T("End");
        COUT << _T("Beam ") << bmVMD.iId << _T(" ") << szEnd << _T(": ")
            << vmd.fVMD[X] << "\t" << vmd.fVMD[Y] << "\t" << vmd.fVMD[Z] << "\t"
            << vmd.fVMD[OX] << "\t" << vmd.fVMD[OY] << "\t" << vmd.fVMD[OZ] << endl;
    } // for(int k = 0; k < bmVMD.vVMD.size(); k++)
    // for(int i = 0; i < nBeams; i++) {
    COUT << endl;
}

```

A few things you should be aware of:

1. A label such as in a material or load case can't contain any spaces. It must be less than 127 characters long.
2. The program checks for input errors (such as duplicate nodes) prior to performing analysis or report. The error messages are listed in the log file that resides in the same directory as the input file. It is always a good idea to check this file even if no errors are reported.
3. It is highly recommended that you study the "Technical Issues" in the Real3D-Analysis program manual, which is freely available from CGI company web site (www.cg-inc.com). You will get a good understanding of the theoretical background on which SolverBlaze is based.

.NET Interface

The .NET interface is a Class Library DLL (cgiSolverBlazeCli.dll). It is written in C++/CLI and makes calls to the native Windows DLL (cgiSolverBlaze.dll). The actual procedure to use this .NET interface is identical to C++ interface, except syntax differences.

Four versions of cgiSolverBlazeCli.dll are provided for your configuration needs: Debug Unicode, Release Unicode, Debug Non-Unicode and Release Non-Unicode. Make sure your project has reference to the correct cgiSolverBlazeCli.dll. Also make sure a copy of libguide40.dll, cgiSolverBlaze.dll, cgiSolverBlazeCli.dll is placed in the exe directory.

Here are the steps to build and run debug version of the test project cgiSolverBlazeTestCSharp in Visual Studio 2005.

1. Download SolverBlaze Library for Visual Studio 2005
2. Download C# SolverBlaze Sample Test (cgiSolverBlazeTestCSharp project)
3. Add Reference to cgiSolverBlazeCli.dll under “distributables\debug”
4. Build cgiSolverBlazeTestCSharp project
5. Copy cgiSolverBlaze.dll, cgiSolverBlazeCli.dll and libguide40.dll under “distributables\debug”
6. Run cgiSolverBlazeTestCSharp test (console application)

The steps to build and run other configurations of the test project cgiSolverBlazeTestCSharp in Visual Studio 2005 are the same as above except changing “distributables\debug” to “distributables\release” etc.