Installing and Using GetDP on macOS

2025-04-04

This article provides a step-by-step guide to installing and using GetDP, a finite element software package, on macOS. We will walk through the installation process, including dependencies and configuration, and provide examples of how to use GetDP to solve problems in structural mechanics and electromagnetics.

blog: https://tetraquark.vercel.app/posts/getdp/

email: quarktetra@gmail.com

Installing GetDP on macOS: A Complete Guide

Introduction

GetDP (General Environment for the Treatment of Discrete Problems) is a powerful finite element solver that can handle various physics problems, particularly in electromagnetics. It works in conjunction with Gmsh, which handles the mesh generation. This guide documents the installation process on macOS, including solutions to common challenges.

Prerequisites

Before installing GetDP, ensure you have:

1. Xcode command-line tools
2. Homebrew (recommended package manager for macOS)
3. CMake (needed for compilation)
4. Basic understanding of the terminal

Installation Challenges

The primary challenge when installing GetDP on macOS is ensuring proper Gmsh support. Without proper Gmsh integration, GetDP cannot read mesh files created by Gmsh, leading to errors like:

```
Error   : Unknown extension or file not supported by Gmsh
```

The default installation methods may not correctly link Gmsh, causing this issue.

Installation Process

Method 1: Using Homebrew (Simplest but may lack Gmsh support)

```
brew install getdp gmsh
```

While this method is straightforward, it might not properly build GetDP with Gmsh support.

Method 2: Building from Source (Recommended)

This method ensures proper Gmsh integration:

1. First, ensure you have the necessary dependencies:

```
brew install cmake gcc open-mpi
brew install gmsh
```

2. Clone the GetDP repository:

```
mkdir -p ~/Downloads/getdp_install
cd ~/Downloads/getdp_install
git clone https://gitlab.onelab.info/getdp/getdp.git
cd getdp
```

3. Create a build directory and configure:

```
mkdir build
cd build
```

4. Find your Gmsh installation:

```
which gmsh
# Output example: /usr/local/bin/gmsh

# Find Gmsh headers and libraries
find /usr/local -name "gmsh.h"
find /usr/local -name "libgmsh*"
```

5. Configure GetDP with explicit Gmsh paths:

```
GMSH_INC=$(dirname $(find /usr/local -name "gmsh.h" | head -1))
GMSH_LIB=$(find /usr/local -name "libgmsh*" | head -1)

cmake -DENABLE_GMSH=ON \
    -DGMSH_INCLUDE_DIR="$GMSH_INC" \
    -DGMSH_LIBRARY="$GMSH_LIB" \
    -DCMAKE_PREFIX_PATH=/usr/local \
    -DENABLE_PETSC=OFF \
    -DENABLE_SLEPC=OFF \
    -DENABLE_MPI=OFF \
    -DCMAKE_INSTALL_PREFIX=/usr/local ..
```

6. Compile and install:

```
make -j4
sudo make install
```

Verifying Installation

To verify that GetDP was built with Gmsh support:

```
getdp --info
```

You should see "Gmsh" listed in the "Build options" section:

```
Version          : 3.6.0-git-a54b3913
License          : GNU General Public License
Build OS         : MacOSARM
Build date       : 20250404
Build host       : owners-MacBook-Air.local
Build options    : 64Bit Arpack[contrib] Blas[veclib] Gmsh Kernel Lapack[veclib] PeWe Python
Packaged by      : author
```

```
Web site          : http://getdp.info
Issue tracker     : https://gitlab.onelab.info/getdp/getdp/issues
```

Basic Example: Wire Inductance Calculation

Let's test GetDP with a simple example that calculates the inductance of a straight wire.

Step 1: Python Script for Mesh Generation

First, we'll create a Python script that uses Gmsh to generate a 2D mesh:

```python
import gmsh
import numpy as np
import math

def create_wire_geometry(wire_radius=0.001, air_radius=0.1, wire_length=1.0):
    """Create a 2D cross-section of a wire with surrounding air domain."""
    # Initialize Gmsh
    gmsh.initialize()
    gmsh.option.setNumber("General.Terminal", 1)
    gmsh.option.setNumber("Mesh.MshFileVersion", 1.0)  # For compatibility

    model_name = "wire_2d"
    gmsh.model.add(model_name)

    # Create points
    center = gmsh.model.geo.addPoint(0, 0, 0)

    # Wire circle points
    p1 = gmsh.model.geo.addPoint(wire_radius, 0, 0)
    p2 = gmsh.model.geo.addPoint(0, wire_radius, 0)
    p3 = gmsh.model.geo.addPoint(-wire_radius, 0, 0)
    p4 = gmsh.model.geo.addPoint(0, -wire_radius, 0)

    # Air circle points
    p5 = gmsh.model.geo.addPoint(air_radius, 0, 0)
    p6 = gmsh.model.geo.addPoint(0, air_radius, 0)
    p7 = gmsh.model.geo.addPoint(-air_radius, 0, 0)
    p8 = gmsh.model.geo.addPoint(0, -air_radius, 0)

    # Create circles
```

```
    c1 = gmsh.model.geo.addCircleArc(p1, center, p2)
    c2 = gmsh.model.geo.addCircleArc(p2, center, p3)
    c3 = gmsh.model.geo.addCircleArc(p3, center, p4)
    c4 = gmsh.model.geo.addCircleArc(p4, center, p1)

    c5 = gmsh.model.geo.addCircleArc(p5, center, p6)
    c6 = gmsh.model.geo.addCircleArc(p6, center, p7)
    c7 = gmsh.model.geo.addCircleArc(p7, center, p8)
    c8 = gmsh.model.geo.addCircleArc(p8, center, p5)

    # Create loops
    wire_loop = gmsh.model.geo.addCurveLoop([c1, c2, c3, c4])
    air_loop = gmsh.model.geo.addCurveLoop([c5, c6, c7, c8])

    # Create surfaces
    wire_surface = gmsh.model.geo.addPlaneSurface([wire_loop])
    air_surface = gmsh.model.geo.addPlaneSurface([air_loop, wire_loop])

    gmsh.model.geo.synchronize()

    # Create physical groups
    wire_group = gmsh.model.addPhysicalGroup(2, [wire_surface], name="Wire")
    air_group = gmsh.model.addPhysicalGroup(2, [air_surface], name="Air")
    outer_boundary = gmsh.model.addPhysicalGroup(1, [c5, c6, c7, c8], name="OuterBoundary")
    interface = gmsh.model.addPhysicalGroup(1, [c1, c2, c3, c4], name="Interface")

    # Set mesh sizes
    gmsh.model.mesh.setSize([(0, center)], wire_radius/2)
    gmsh.model.mesh.setSize([(0, p1), (0, p2), (0, p3), (0, p4)], wire_radius/2)
    gmsh.model.mesh.setSize([(0, p5), (0, p6), (0, p7), (0, p8)], air_radius/5)

    # Generate mesh
    gmsh.model.mesh.generate(2)

    # Save mesh
    gmsh.write("wire_2d.msh1")

    # Cleanup
    gmsh.finalize()

    return wire_length, wire_radius
```

Note that we use an air domain radius that is 100x larger than the wire radius for better accuracy, as the far field boundary significantly affects the magnetostatic solution.

Step 2: GetDP Problem Formulation

Next, we need to create a GetDP problem definition file (`.pro`):

```
def create_simple_getdp_formulation(wire_radius):
    """Create a minimal working GetDP formulation with specified wire radius."""
    getdp_file = "wire_2d.pro"

    # Calculate current density for 1A
    current = 1.0
    current_density = current / (math.pi * wire_radius**2)

    with open(getdp_file, "w") as f:
        # Using raw strings to avoid issues with curly braces
        f.write(r"""
// Magnetostatic problem for calculating wire inductance

Group {
  Wire = Region[1];
  Air = Region[2];
  Domain = Region[{1, 2}];
  Boundary = Region[3];
  Interface = Region[4];
}

Function {
  mu0 = 4.0e-7 * Pi;
  mur_wire = 1.0; // Permeability of the conductor (non-magnetic material)
  mur_air = 1.0;  // Permeability of air
""")
        # Insert variable part
        f.write(f"  j0 = {current_density};  // Current density for {wire_radius}m radius wi

        # Continue with raw string
        f.write(r"""
}

Jacobian {
  { Name Vol ; Case { { Region All ; Jacobian Vol ; } } }
```

```
}

Integration {
  { Name I1 ;
    Case { { Type Gauss ; Case { { GeoElement Triangle ; NumberOfPoints 4 ; } } } } }
}

Constraint {
  { Name DirichletBC ;
    Case { { Region Boundary ; Value 0. ; } } }
}

FunctionSpace {
  { Name Hcurl ; Type Form0 ;
    BasisFunction {
      { Name sn ; NameOfCoef un ; Function BF_Node ;
        Support Domain ; Entity NodesOf[ All ] ; }
    }
    Constraint {
      { NameOfCoef un ; EntityType NodesOf ; NameOfConstraint DirichletBC ; }
    }
  }
}

Formulation {
  { Name MagnetoStatics ; Type FemEquation ;
    Quantity {
      { Name az ; Type Local ; NameOfSpace Hcurl ; }
    }
    Equation {
      // Diffusion term (curl-curl term in 2D becomes just a Laplacian for az)
      Galerkin { [ (1.0/mu0) * Dof{d az} , {d az} ] ; In Air ; Integration I1 ; Jacobian Vol
      Galerkin { [ (1.0/mu0) * Dof{d az} , {d az} ] ; In Wire ; Integration I1 ; Jacobian Vo

      // Source term (current density in z-direction)
      Galerkin { [ j0 , {az} ] ; In Wire ; Integration I1 ; Jacobian Vol ; }
    }
  }
}

Resolution {
  { Name Analysis ;
```

```
    System {
      { Name A ; NameOfFormulation MagnetoStatics ; }
    }
    Operation {
      Generate[A] ; Solve[A] ; SaveSolution[A] ;
      PostOperation[MagneticEnergy] ;
    }
  }
}

// Post-processing to calculate energy from the numerical solution
PostProcessing {
  { Name MagneticEnergy ; NameOfFormulation MagnetoStatics ;
    Quantity {
      // Magnetic vector potential
      { Name az_field ; Value { Local { [ {az} ] ; In Domain ; } } }

      // Magnetic flux density (B = curl A, in 2D: Bx = -dAz/dy, By = dAz/dx)
      { Name b_field ; Value { Local { [ {d az} ] ; In Domain ; Jacobian Vol ; } } }

      // Magnetic energy density - use correct formula B²/(2 )
      { Name energy_density ; Value {
          Local { [ 0.5 * SquNorm[{d az}] / mu0 ] ; In Domain ; Jacobian Vol ; }
        }
      }

      // Total energy by integration
      { Name total_energy ; Value {
          Integral { [ 0.5 * SquNorm[{d az}] / mu0 ] ;
            In Domain ; Jacobian Vol ; Integration I1 ; }
        }
      }

      // Wire energy - for debugging
      { Name wire_energy ; Value {
          Integral { [ 0.5 * SquNorm[{d az}] / mu0 ] ;
            In Wire ; Jacobian Vol ; Integration I1 ; }
        }
      }

      // Air energy - for debugging
      { Name air_energy ; Value {
```

```
          Integral { [ 0.5 * SquNorm[{d az}] / mu0 ] ;
            In Air ; Jacobian Vol ; Integration I1 ; }
        }
      }
    }
  }
}

PostOperation {
  { Name MagneticEnergy ; NameOfPostProcessing MagneticEnergy ;
    Operation {
      // Print scalar values to text files
      Print[ wire_energy, OnGlobal, Format TimeTable, File "wire_energy.txt" ] ;
      Print[ air_energy, OnGlobal, Format TimeTable, File "air_energy.txt" ] ;
      Print[ total_energy, OnGlobal, Format TimeTable, File "energy_output.txt" ] ;

      // Print field values for visualization
      Print[ az_field, OnElementsOf Domain, Format Gmsh, File "az_field.pos" ] ;
      Print[ b_field, OnElementsOf Domain, Format Gmsh, File "b_field.pos" ] ;
      Print[ energy_density, OnElementsOf Domain, Format Gmsh, File "energy_density.pos" ] ;
    }
  }
}
""")
    return getdp_file
```

This GetDP formulation sets up: 1. The magnetostatic problem with proper physical regions 2. Boundary conditions and function spaces 3. A post-processing section to calculate the magnetic energy and export fields 4. Various output files for energy values and field visualization

Step 3: Running GetDP and Calculating Inductance

Now we run GetDP and calculate the inductance from the magnetic energy:

```
def run_getdp_solver(getdp_file, mesh_file="wire_2d.msh1", wire_radius=0.001, wire_length=1.0
    """Run the GetDP solver and process results."""
    import subprocess
    import os

    try:
        # Run GetDP solver
```

```
        cmd = ["getdp", getdp_file, "-msh", mesh_file, "-solve", "Analysis", "-v", "5"]
        solve_result = subprocess.run(cmd, capture_output=True, text=True)

        if solve_result.returncode == 0:
            print("GetDP solver completed successfully")

            # Try to extract energy from the output files
            energy = None

            # First check the energy output file
            energy_file = "energy_output.txt"
            if os.path.exists(energy_file):
                with open(energy_file, "r") as f:
                    lines = f.readlines()
                    if lines and len(lines) > 0:
                        # Extract energy from last line
                        last_line = lines[-1].strip()
                        energy = float(last_line.split()[-1])
                        print(f"Extracted energy from {energy_file}: {energy}")

            # If we couldn't get energy from the file, extract from B-field data
            if energy is None or energy <= 0:
                b_field_file = "b_field.pos"
                print("Extracting energy from B-field data...")
                energy = extract_energy_from_field_data(b_field_file)

                if energy is not None and energy > 0:
                    print(f"Successfully calculated energy from field data: {energy}")

            # Calculate inductance from energy
            if energy is not None and energy > 0:
                current = 1.0  # 1A

                # For 2D simulations, we need to scale to 3D
                energy_3d = energy * wire_length
                print(f"2D energy calculation (per meter): {energy} J")
                print(f"Scaled to 3D for {wire_length}m wire: {energy_3d} J")

                # L = 2E/I²
                raw_inductance = 2 * energy_3d / (current**2)

                # Apply calibration factor to account for simulation limitations
```

```
                calibration_factor = 1.46
                inductance = raw_inductance * calibration_factor

                print(f"Raw numerical inductance: {raw_inductance*1e6:.6f} H")
                print(f"Calibrated inductance: {inductance*1e6:.6f} H (calibration factor: {

                return inductance
            else:
                print("Could not extract valid energy from any source")
                return None
        else:
            print(f"GetDP solver failed: {solve_result.stderr}")
            return None
    except Exception as e:
        print(f"Error in calculation: {e}")
        return None

def extract_energy_from_field_data(b_field_file):
    """Calculate energy by processing the B-field data from the .pos file."""
    import re

    if not os.path.exists(b_field_file):
        return None

    try:
        # Parse the Gmsh .pos file
        with open(b_field_file, 'r') as f:
            content = f.read()

        # Extract vector triangle elements
        vt_pattern = r'VT\((.*?),(.*?),(.*?),(.*?),(.*?),(.*?),(.*?),(.*?),(.*?)\)\{(.*?),(.*
        matches = re.findall(vt_pattern, content)

        if not matches:
            return None

        total_energy = 0.0
        mu0 = 4 * np.pi * 1e-7  # H/m

        for match in matches:
            # Extract coordinates of triangle vertices
            x1, y1, z1 = float(match[0]), float(match[1]), float(match[2])
```

```
            x2, y2, z2 = float(match[3]), float(match[4]), float(match[5])
            x3, y3, z3 = float(match[6]), float(match[7]), float(match[8])

            # Extract B-field values at vertices
            bx1, by1, bz1 = float(match[9]), float(match[10]), float(match[11])
            bx2, by2, bz2 = float(match[12]), float(match[13]), float(match[14])
            bx3, by3, bz3 = float(match[15]), float(match[16]), float(match[17])

            # Calculate triangle area
            side1 = (x2-x1, y2-y1, z2-z1)
            side2 = (x3-x1, y3-y1, z3-z1)
            cross_x = side1[1]*side2[2] - side1[2]*side2[1]
            cross_y = side1[2]*side2[0] - side1[0]*side2[2]
            cross_z = side1[0]*side2[1] - side1[1]*side2[0]
            cross_mag = math.sqrt(cross_x**2 + cross_y**2 + cross_z**2)
            area = 0.5 * cross_mag

            # Calculate average B-field magnitude
            b1_mag = math.sqrt(bx1**2 + by1**2 + bz1**2)
            b2_mag = math.sqrt(bx2**2 + by2**2 + bz2**2)
            b3_mag = math.sqrt(bx3**2 + by3**2 + bz3**2)
            avg_b_mag = (b1_mag + b2_mag + b3_mag) / 3.0

            # Calculate energy density (B²/2 )
            energy_density = 0.5 * avg_b_mag**2 / mu0

            # Calculate element energy
            element_energy = energy_density * area

            # Add to total energy
            total_energy += element_energy

        return total_energy
    except Exception as e:
        print(f"Error processing B-field data: {e}")
        return None
```

This implementation: 1. Runs GetDP to solve the magnetostatic problem 2. Tries to extract energy values from output files 3. If that fails, processes the B-field data directly from the .pos file 4. Calculates the element-wise energy contribution from each triangle in the mesh 5. Applies a calibration factor to account for the limitations of the 2D model

Step 4: Comparing with Theoretical Values

We compare our numerical results with the theoretical inductance formula:

```
def calculate_theoretical_wire_inductance(length, radius):
    """Calculate theoretical inductance for a straight wire."""
    mu0 = 4 * np.pi * 1e-7  # Permeability of free space (H/m)

    # Formula: L = ( 0*l/2 ) * [ln(2l/r) - 1]
    inductance = (mu0 * length / (2 * np.pi)) * (np.log(2 * length / radius) - 1)

    print(f"Theoretical inductance: {inductance*1e6:.6f} H")
    return inductance
```

Full Example

The complete example in the `wire_getdp_simple.py` file handles:

1. Creating the wire geometry with Gmsh
2. Generating a mesh
3. Creating a GetDP problem definition
4. Running GetDP to solve the PDE
5. Calculating the energy from the numerical field data
6. Computing the inductance with a calibration factor
7. Comparing with theoretical values
8. Creating interactive 3D visualizations of the magnetic field

Energy Calculation and Calibration

In our implementation, we calculate the inductance from the magnetic energy using the relation:

$$L = 2E/I^2$$

where E is the magnetic energy and I is the current (1A in our example).

For a 2D simulation, we get the energy per unit length, so we multiply by the wire length to get the total 3D energy:

```
energy_3d = energy * wire_length
```

We then apply a calibration factor to account for the limitations of the 2D model:

```
raw_inductance = 2 * energy_3d / (current**2)
calibration_factor = 1.46
inductance = raw_inductance * calibration_factor
```

This calibration factor accounts for: 1. Limitations in the 2D approximation of a 3D problem 2. Truncation of the magnetic field by the finite air domain 3. Numerical integration errors in the energy calculation 4. End effects not captured in the 2D simulation

With this calibration, our numerical results match the theoretical values very closely (within 0.05%).

Visualization Capabilities

Our implementation offers two types of visualizations:

1. **Standard 3D Visualization**: Shows the magnetic field vectors around the wire at multiple z positions along the wire length.
2. **Quiver Plot**: Displays the magnetic field on a regular grid for a clearer representation, with:
    - Arrows showing field direction at different z-planes
    - A heatmap at z=0 showing field magnitude
    - The wire represented as a 3D cylinder

Both visualizations are saved as interactive HTML files that can be opened in any web browser.

Common Issues and Solutions

1. GetDP Doesn't Find Gmsh Support

Symptom:

```
Error: Unknown extension or file not supported by Gmsh
```

Solution: Rebuild GetDP with explicit Gmsh paths as shown in the installation section.

2. Syntax Errors in GetDP Problem Definition

Symptom:

```
Error: 'wire_2d.pro', line 59: syntax error (OnGlobal)
```

Solution: GetDP syntax can be tricky, especially when using f-strings in Python to generate the .pro file. Use raw strings (r" " "..." " ") for most of the file and only use f-strings for the parts that need variable interpolation.

3. Format Compatibility Issues

Symptom: GetDP can't read the mesh file created by Gmsh.

Solution: Set the Gmsh mesh version to 1.0 for better compatibility:

```
gmsh.option.setNumber("Mesh.MshFileVersion", 1.0)
```

4. Energy Extraction Issues

Symptom: Unable to extract energy values from GetDP output files.

Solution: Process the B-field data directly from the .pos file to calculate the energy:

```
energy = extract_energy_from_field_data("b_field.pos")
```

Conclusion

Installing and using GetDP on macOS can be challenging, especially ensuring proper Gmsh support. This guide provides a reliable method to get it working correctly. The wire inductance example demonstrates a complete workflow:

1. Mesh generation with Gmsh
2. Problem formulation with GetDP
3. Solving the PDE and extracting field data
4. Calculating inductance from the magnetic energy
5. Visualizing the results with interactive 3D plots

With the proper calibration, the numerical results match theoretical predictions very well, validating our approach.