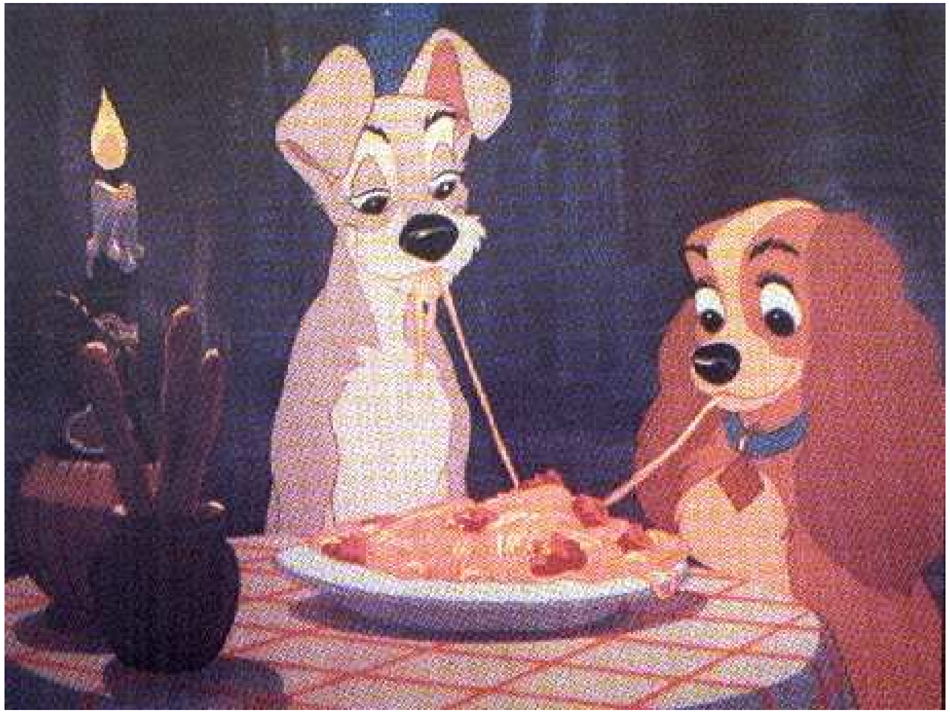
A First Step Towards Automatic PDE Code Verification

Ola Skavhaug
Kent-Andre Mardal
Hans Petter Langtangen

Simula Research Laboratory, Norway
University of Oslo, Norway



Validation and Verification

Validation:

- **Is the PDE model appropriate?**
- **Or: Do we solve the right equations?**
- **Core interest among scientists and engineers**

Verification:

- **Are the numerical methods correctly implemented?**
- **Or: Do we solve the equations right?**
- **Attracts much less interest than validation**
- **Validation requires successful verification**

Generation of Analytical Solutions

Given a PDE or system of PDEs:

$$F(u) = 0 \textbf{ in } \Omega$$

(F **and** u **may be scalar or vector)**

Pick any v **as analytical solution**

v **solves the problem**

$$F(u) = b \textbf{ in } \Omega, \quad b = F(v)$$

$$\textbf{e.g. } u = v \textbf{ on } \partial\Omega$$

i.e., add a source term in the equation such that v **is a solution**

Known as the *method of manufactured solutions*

Verification Procedure

Make a sequence of refinements in space and time

Compute errors

Fit a convergence estimate

$$\textbf{error} = A\Delta x^p + B\Delta y^q + C\Delta z^r + D\Delta t^s$$

(by nonlinear least squares)

Does the method converge? With expected rate?

This is easy, but not a standard or required scientific procedure in Computational Science

Computing $F(v)$ (Source Term)

Computing the source term $F(v)$ is a matter of differentiation

Calculating $F(v)$ by hand is tedious and error-prone

This is critical for coupled systems of PDEs

Example

$$
\begin{aligned}
\nabla \cdot [\mu \nabla w] &= -\beta \\
\nabla^2 T &= -\kappa^{-1} \mu \dot{\gamma}^2
\end{aligned}
$$

$$
\begin{aligned}
\mu &= e^{-\alpha(T - T_0)} \dot{\gamma}^{n-1} \\
\dot{\gamma} &= \sqrt{(w_{,x})^2 + (w_{,y})^2}
\end{aligned}
$$

$$
\begin{aligned}
w &= \cos(\sin(x + y)) + \tan x \\
T &= x + y + e^{xy}
\end{aligned}
$$

Example cont.

```
F1:
exp(y*x)*x**2+y**2*exp(y*x)+(10.0)*((1-sin(sin(y+x))*cos(y+x)+tan(x)**2)**2+
(-sin(sin(y+x))*cos(y+x))**2)*(0.1+3*((1-sin(sin(y+x))*cos(y+x)+tan(x)**2)**2+
(-sin(sin(y+x))*cos(y+x))**2)**(-0.275))*exp(0.05-0.5*y-0.5*exp(y*x)-0.5*x)

F2:
0.1-cos(sin(y+x))*cos(y+x)**2*(0.1+3*((1-sin(sin(y+x))*cos(y+x)+
tan(x)**2)**2+(-sin(sin(y+x))*cos(y+x))**2)**(-0.275))*exp(0.05-0.5*y-
0.5*exp(y*x)-0.5*x)-(0.825)*((1-sin(sin(y+x))*cos(y+x)+tan(x)**2)**2+
(-sin(sin(y+x))*cos(y+x))**2)**(-1.275)*(-(2.0)*sin(sin(y+x))*cos(y+x)*
(-cos(sin(y+x))*cos(y+x)**2+sin(sin(y+x))*sin(y+x))+(2.0)*(1-sin(sin(y+x))*
cos(y+x)+tan(x)**2)*((2.0)*tan(x)*(1+tan(x)**2)-cos(sin(y+x))*cos(y+x)**2+
sin(sin(y+x))*sin(y+x)))*(1-sin(sin(y+x))*cos(y+x)+tan(x)**2)*exp(0.05-0.5*y-
0.5*exp(y*x)-0.5*x)+(0.825)*((1-sin(sin(y+x))*cos(y+x)+tan(x)**2)**2+
(-sin(sin(y+x))*cos(y+x))**2)**(-1.275)*sin(sin(y+x))*cos(y+x)*((2.0)*
(1-sin(sin(y+x))*cos(y+x)+tan(x)**2)*(-cos(sin(y+x))*cos(y+x)**2+
sin(sin(y+x))*sin(y+x))-2*sin(sin(y+x))*cos(y+x)*(-cos(sin(y+x))*cos(y+x)**2+
sin(sin(y+x))*sin(y+x)))*exp(0.05-0.5*y-0.5*exp(y*x)-0.5*x)-
(-0.5-0.5*exp(y*x)*x)*sin(sin(y+x))*cos(y+x)*(0.1+3*((1-sin(sin(y+x))*
cos(y+x)+tan(x)**2)**2+(-sin(sin(y+x))*cos(y+x))**2)**(-0.275))*
exp(0.05-0.5*y-0.5*exp(y*x)-0.5*x)+(-0.5-0.5*y*exp(y*x))*(1-sin(sin(y+x))*
```

Automation Idea 1

Use Maple (or similar) to define $F(u)$ and v. Then compute $F(v)$ and generate C/F77 code

Incorporate this C/F77 code in the simulator.

Easy in principle, but still somewhat tedious; requires some manual work

The process must be simpler and safer if we want to make *extensive* use of manufactured solutions!

Automation Idea 2

Build a fully automatic "problem solving environment" for this type of code verification

Need to glue symbolic package, source term $F(v)$ code generation, and PDE solver

Use a "scripting language" for gluing!
– we have chosen Python

Our Approach

GiNaC: symbolic math engine (in C++)

Generate Python interface to GiNaC (using SWIG)

Extend Python-GiNaC with (e.g.) `grad` and `div`

Diffpack: PDE solver library (in C++)

Generate Python interface to Diffpack PDE solver (using SWIG/SIP)

Famms: home-made Python module to glue GiNaC and solver; specify $F(u)$ and v in Python, the rest is automatic

Example

$$\nabla\left((\lambda+\mu)\nabla\cdot\boldsymbol{u}\right)+\nabla\cdot(\mu\nabla\boldsymbol{u})=\mathbf{0}$$

```
from Elasticity1MG import *   # Diffpack PDE solver
el = Elasticity1MG()          # PDE solver as C++/Python object
from famms import *
from symbolic import *

f = Famms(nspacedim=2)
x, y = f.x                   # aliases for independent variables
v1 = sin(x); v2 = cos(y)     # pick functions for each component
v = Vector((x,y), (v1,v2))   # specify manufactured solution

Lambda = 120; mu = 3         # elasticity parameters

def F(u):
    return grad((Lambda+mu)*div(u)) + div(mu*grad(u))

f.assign(equation=F, solution=v, simulator=el)
```

Example cont.

$$\nabla\left((\lambda+\mu)\nabla\cdot\boldsymbol{u}\right)+\nabla\cdot(\mu\nabla\boldsymbol{u})=\mathbf{0}$$

```
# assign input data:
m = MenuSystem()                    # menu interface to PDE solver
el.define(m)                        # define data structs for input
m.set("multilevel method", "Multigrid")
m.set("smoother basic method", "SSOR")
m.set("coarse grid basic method", "GaussElim")
m.set("Lambda", Lambda); m.set("mu", mu)
...
el.scan()                           # send input to solver

el.solveProblem()

el.saveResults()
```

Another Example

$$
\begin{aligned}
\nabla \cdot [\mu \nabla w] &= -\beta \\
\nabla^2 T &= -\kappa^{-1} \mu \dot{\gamma}^2 \\
\mu &= e^{-\alpha(T - T_0)} \dot{\gamma}^{n-1} \\
\dot{\gamma} &= \sqrt{(w_{,x})^2 + (w_{,y})^2}
\end{aligned}
$$

```
w = cos(sin(x+y))+tan(x)
T = x+y+exp(x*y)

def gamma(w):      return sqrt(w.diff(x,1)**2+w.diff(y,1)**2)
def my_w(gamma):   return gamma**(n-1)
def my_T(T):       return exp(-alpha*(T-T0))
def my(T,gamma):   return my_T(T)*my_w(gamma)

def F1(w,T): return laplace(T)+kappa**(-1)*my(T,gamma(w))*gamma(w)**2
def F2(w,T): return div(my(T,gamma(w))*grad(w))+beta
```

What Have We Done?

We have

- **“created” an engine for symbolic PDE math, with nice input syntax**
- **made a seamless integration of our native PDE solver and this math engine such that the PDE solver solves a *new* symbolically computed problem – no solver modification required**
- **incorporated empirical convergence estimation**
- **incorporated visualization (Vtk/MayaVi)**

...and this works for any Diffpack PDE solver

Beyond Diffpack

How much of this is actually tied to Diffpack?

Famms is meant to be independent of the PDE package

Key mechanism: determine functions run–time

Functors (function objects) in C++

Function pointers in plain C

Should be “easy” to incorporate in other software packages; we only need Python callback mechanisms

The Glue

v and $F(v)$ are expressions with GiNaC objects

```
v = sin(x*y)      # x, y, v: GiNaC objects
```

Easy to evaluate v pointwise in GiNaC

Bind generic Python pointers in the PDE solver to Python functions for evaluating v and source term $F(v)$

```
class MyRHS : public BaseClassFunctor
{
public:
  PyObject*  pyfunc;  // Python function to be called

  double operator() (const spacepoint& x, time t)  {
    PyObject*  args;    // arguments to pyfunc
    // build args from x and t
    pyresult = PyEval_CallObject (pyfunc, args);
    double_result = PyFloat_AsDouble(pyresult);
    return double_result;
  }
```