Symbolic Methods

I. Hawke

Mathematical Sciences,
University of Southampton, UK

Advanced Computational Methods 1, Semester 1

Complexity

Symbolic language encodes the model, or (approximate) solution.

Explicit symbolic form often

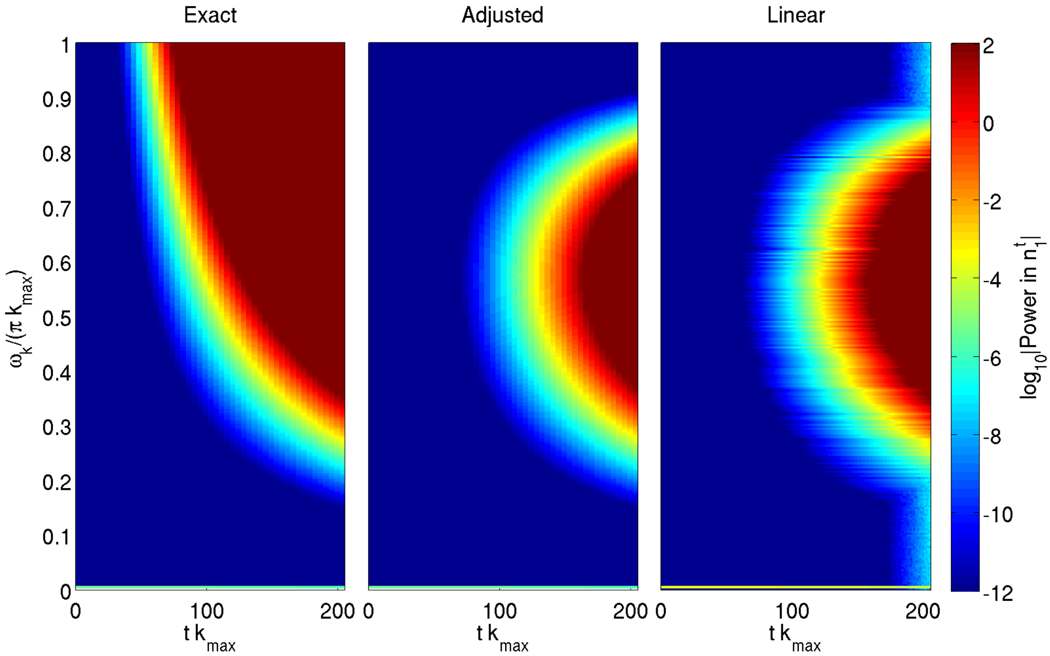
- lengthy, but
- derived from compact, abstract form.

Use symbolic algebra methods to go directly from high-level, compact form to explicit code. This

- saves time;
- reduces errors;
- is easier for humans to read;
- allows high-level optimization.

Two stream problem: statement

University of Southampton
School of Mathematics



A nonlinear, relativistic fluid instability problem. Physical instability or bug? Direct comparison with linearized solution shows it's correct.

Two stream problem: solution

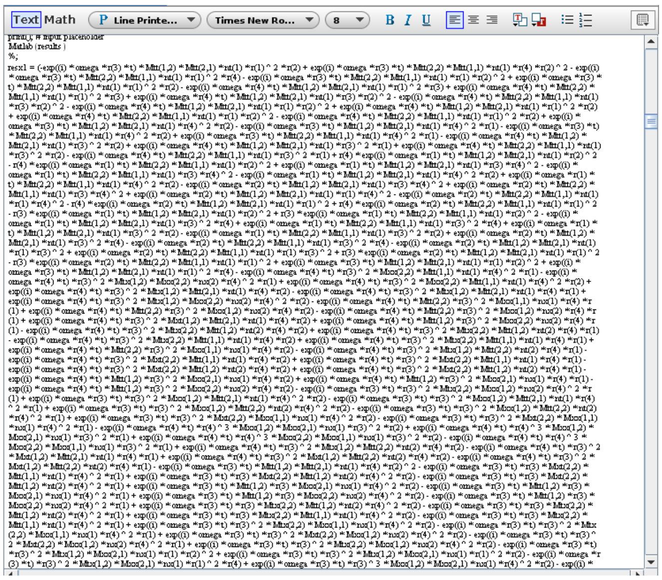
University of Southampton
School of Mathematics

Solution via Fourier series ($x \to \omega$) and Laplace transforms ($t \to s$). Gives 2×2 linear problem

$$A(s)\,\delta\tilde{n} = b(s)$$

for the unknown perturbation $\delta\tilde{n}$. Solve; inverse Laplace transform gives $\delta n(t)$ for given frequency.

Actual explicit solution fills *seven* screens for one term.



Symbolic methods

Symbolic methods

- work with symbols, not numbers;
- rely on pattern matching;
- often require making mathematical assumptions explicit.

We'll use `sympy` as an example system; alternatives include

- Mathematica (Wolfram Alpha)
- Maple
- Sage MathCloud
- etc.

Basics

`sympy` is another python module. Standard `import` to get going.

Need to *explicitly define* symbols:

```
In []:  x, y = sympy.symbols("x, y")
```

Can then create new symbols and manipulate existing ones:

```
In []:  z = x**2 + x*y
In []:  sympy.diff(z, x)
Out []:  2*x + y
In []:  sympy.integrate(z, y)
Out []:  x**2*y + x*y**2/2
```

Types

Three basic built-in types.

1. Symbols: `sympy.Symbol` or `sympy.symbols`. A single unknown (or unknowns) as above.
2. Functions: `sympy.Function`. A single (undefined) function.
3. Matrices: `sympy.Matrix`. A single matrix with given size and entries.

Assumptions

Need to think carefully about assumptions. For example:

```
In []:  f = sympy.Function("f")
In []:  sympy.diff(f(x), x)
Out []:  Derivative(f(x), x)
In []:  sympy.diff(f(x+y), x)
Out []:  Subs(Derivative(f(_xi_1), _xi_1), (_xi_1,), 
(x + y,))
```

Think carefully to see that this means

$$\left.\frac{d\,f(\xi_1)}{d\,\xi_1}\right|_{\xi_1=x+y}.$$

Explicit functions

`sympy` reimplements all the important functions symbolically: use those, *not* e.g. `numpy` or `math`.

```
In []:  sympy.integrate(1 + x**2*sympy.exp(x) +
sympy.log(x**2+1), x)
Out []:  x*log(x**2 + 1) - x + (x**2 - 2*x +
2)*exp(x) + 2*atan(x)
```

When using internal functions, `sympy` can `simplify`:

```
In []:  sympy.simplify(sympy.sin(x)**2 +
sympy.cos(x)**2)
Out []:  1
```


Linear Algebra

Can construct and solve symbolic or numerical matrix problems.

```
In []:  a,b,c,d,e,f = sympy.symbols("a,b,c,d,e,f")
In []:  A = sympy.Matrix([[a,b],[c,d]])
In []:  rhs = sympy.Matrix([[e],[f]])
In []:  A.solve(rhs)
Out []:  Matrix([
[e*(1/a + b*c/(a**2*(d - b*c/a))) - b*f/(a*(d -
b*c/a))],
[ f/(d - b*c/a) - c*e/(a*(d - b*c/a))]])
```

Code generation

Key objective: convert high level mathematics to code. Two methods:

1. `lambdify`: construct a python function to use immediately.
2. `codegen`: produce source code (C/Fortran/...) to use later.

Example: given

$$\phi(x, t) = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right)$$

write a function evaluating ϕ' given the parameters.

lambdify

$$\phi(x, t) = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right);$$

write a function evaluating ϕ' given the parameters.

```
In []: from sympy.utilities.lambdify import lambdify
In []: x, nu, t = sympy.symbols("x nu t")
In []: phi = sympy.exp(-(x-4*t)**2/(4*nu*(t+1))) + \
       sympy.exp(-(x-4*t-2*sympy.pi)**2/(4*nu*(t+1)))
In []: phiprime = phi.diff(x)
In []: dphi = lambdify((t, x, nu), phiprime)
In []: print("Derivative at t=1, x=4, nu=3 is {}".\
       format(dphi(1.0, 4.0, 3.0)))
Out []: Derivative at t=1, x=4, nu=3 is
        0.10106780505315822
```

$$\phi(x,t) = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right);$$

write a function evaluating ϕ' given the parameters.

```
In []: from sympy.utilities.codegen import codegen
In []: x, nu, t = sympy.symbols("x nu t")
In []: phi = sympy.exp(-(x-4*t)**2/(4*nu*(t+1))) + \
       sympy.exp(-(x-4*t-2*sympy.pi)**2/(4*nu*(t+1)))
In []: phiprime = phi.diff(x)
In []: code = codegen(('my_derivative', phiprime), 'C', \
       'my_project')
```


Results of `codegen` include C code and header file:

```
In []: print(code[0][1])
Out []:
/******************************************************************************
 *                      Code generated with sympy 0.7.5                      *
 *                                                                            *
 *              See http://www.sympy.org/ for more information.              *
 *                                                                            *
 *                       This file is part of 'project'                      *
 ******************************************************************************/
#include "my_project.h"
#include <math.h>

double my_derivative(double nu, double t, double x) {

   return -1.0L/4.0L*(-8*t+2*x)*exp(-1.0L/4.0L*pow(-4*t+x,2)/(nu*(t+1)))/(nu*(t+1)) \
          -1.0L/4.0L*(-8*t+2*x-4*M_PI)*exp(-1.0L/4.0L*pow(-4*t+x-2*M_PI,2)/(nu*(t+1))) \
          /(nu*(t+1));

}
In []: print(code[1][1])
Out []:
#ifndef PROJECT__MY_PROJECT__H
#define PROJECT__MY_PROJECT__H

double my_derivative(double nu, double t, double x);

#endif
```

Summary

- Symbolic methods save time and reduce bugs when producing code.
- Careful attention to assumptions must be paid.
- Explicit definition of key symbols is required.
- Code generation to scripted or compiled language possible.

In the lab:

- start from Taylor series for arbitrary function,
- use `sympy`, generate central difference approximation to first derivative at arbitrary order,
- check convergence of scripted and compiled code.

Note: use `sympy.init_printing()` to get nicer screen output.