MANCHESTER
1824

The University
of Manchester

iGen: A program for the automated generation of models and parameterisations

Tang, D.F. and Dobbie, S.

2011

MIMS EPrint: **2011.25**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Manuscript prepared for Geosci. Model Dev.
with version 3.2 of the LaTeX class copernicus.cls.
Date: 25 March 2011

iGen: A program for the automated generation of models and parameterisations

D. F. Tang[1] **and S. Dobbie**[2]

[1]Manchester Institute for Mathematical Sciences, University of Manchester, Oxford Rd., Manchester, UK.
[2]School of Earth and Environment, University of Leeds, Leeds, UK.

Abstract. Complex physical systems can often be simulated using very high-resolution models but this is not always practical because of computational restrictions. In this case the model must be simplified or parameterised, but this is a notoriously difficult process that often requires the introduction of 'model assumptions' that are hard or impossible to justify. Here we introduce a new approach to parameterising models. The approach makes use of a newly developed computer program, which we call iGen, that analyses the source code of a high-resolution model and formally derives a much faster parameterisation that closely approximates the original, reporting bounds on the error introduced by any approximations. These error bounds can be used to formally justify use of the parameterised model in subsequent numerical experiments. Using increasingly complex physical systems as examples we illustrate that iGen has the ability to produce parameterisations that run typically orders of magnitude faster than the underlying, high-resolution models from which they are derived and show that iGen has the potential to become an important tool in model development.

1 Introduction

The process of creating parameterisations for geophysical models often requires the introduction of "model assumptions" and it is generally accepted that these assumptions are often difficult to justify from the underlying equations of motion (Winsberg, 2001). Usually the physical process to be parameterised can quite easily be simulated accurately using a sufficiently high-resolution model that resolves all the relevant scales, this model can be justified since by definition it integrates the underlying equations of motion with sufficient accuracy. However, superparameterisations aside (Khairoutdinov et al., 2005), such high-resolution models would run far too slowly to be considered to be practical parameterisation schemes. However, suppose we view the high-resolution model as a formal definition of the desired behaviour of the parameterisation, rather than a code to be executed. The problem of parameterisation then reduces to one of finding a computer program that closely approximates the high-resolution model but uses far fewer computational operations. To this end, we have developed iGen: a program that analyses the source code of a high-resolution model, applies appropriate approximations, derives the source code of a faster model and reports bounds on the error between the fast model and the high-resolution model. Because the parameterisation is formally derived from the high-resolution model, the output from the parameterisation can be used to justify conclusions about the behaviour of the high-resolution model, and so about the physical system of interest.

iGen can be used to generate parameterisations in the following way: We begin with a high-resolution model of the physical process to be parameterised. The inputs and outputs of this model will be the high-resolution start and end state of the simulation. For the parameterisation, however, we require less detailed inputs and outputs so we must 'wrap' the model in some extra code to account for this. The exact way this is done depends somewhat on the nature of the parameterisation. However, in general, we add code at the end of the model to extract the value of the desired physical quantity from the high-resolution output, this is usually straight forward as it involves just throwing some information away. In addition, we add code at the beginning to transform the input to the parameterisation into the high-resolution input required by the model. This is generally more involved since the input to the parameterisation may not uniquely specify a high-resolution state, in this case we use a random number generator to create a distribution of possible high-resolution states. This distribution represents the conditional probability of being in a specific high-resolution state given the

Correspondence to: D. F. Tang
(daniel.tang@manchester.ac.uk)

```
input(T)
  // Wrapper: turn T into position & velocity
  // Rand() returns a random number in [-1:1]
  x = (Rand()+1.0)/2.0
  y = (Rand()+1.0)/2.0
  angle = PI*(Rand()+1.0)
  speed = sqrt(T + T*Rand()/2.0)
  vx = speed*sin(angle)
  vy = speed*cos(angle)

  // Simulate atom bouncing around box
  p = 0.0 // pressure (impulse)
  t = 0.0 // time
  while(t < TMAX) {
    x = x + vx*DT
    y = y + vy*DT
    if(x > 1.0) {
      x  = 2.0 - x
      p = p + (2.0 * vx)
      vx = -vx
    }
    if(x < 0.0) {
      x  = -x
      vx = -vx
    }
    if(y > 1.0) {
      y  = 2.0 - y
      vy = -vy
    }
    if(y < 0.0) {
      y  = -y
      vy = -vy
    }
    t = t + DT
  }

  // Wrapper: return average impulse per sec
  p = p/TMAX
output(p)
```

Fig. 1. Program to simulate an atom bouncing around a 2-dimensional box

input state of the parameterisation. This 'wrapped' model now has the required inputs and outputs and can be analysed by iGen for a user-specified range of inputs. iGen can then 'integrate over' any random numbers and apply approximations to generate an efficient parameterisation scheme with formally bounded error in means and, if desired, higher moments.

To illustrate this technique consider a simulation of a gas contained within a 2-dimensional box and suppose that we wish to parameterise the pressure of the gas in terms of its temperature. In this case, the high-resolution model is a simulation of an atom bouncing around a box of unit dimension. To wrap the model we extract the pressure by calculating the average impulse per second on the right hand wall of the box and use this as output. The input is the temperature which is proportional to the average kinetic energy of the atom (for simplicity we assume a top hat distribution of kinetic energy rather than the Maxwell-Boltzmann distribution, but this does not affect the result). The initial position and direction of motion of the atom is chosen at random using a random number generator. The pseudocode for the program is shown in figure 1 with the different sections marked.

iGen was used to analyse this program. The analysis required no approximations (other than the finite precision of the `sqrt`, `sin` and `cos` functions) and after integrating over the random numbers, the result was a successful identification of the thermodynamic relationship, $p \propto T$, as we would expect from the theory of ideal gasses.

2 Symbolic analysis

iGen works by using the technique of 'symbolic analysis' (Fahringer and Scholz, 2003) in which the variables of the program are not considered to be floating point numbers with specific values but instead are considered to be symbolic expressions that are functions of the input variables. iGen considers variables to be pairs (C,b) where C is a multivariate polynomial and b is a constant bound on the error between C and the value the variable would have in the absence of any approximations. We call these pairs 'polynomial bounds'.

For example, suppose we wish to generate an approximation of the very simple program

```
input(x)
  a = x + 1.0
  y = a*a
  y = y*y
output(y)
```

for inputs in the range $-0.1 \leq x \leq 0.1$.

Normally, we would simply execute this program for some input, say $x = 0.1$. So, after the first line $a = 1.1$, after the second line $y = 1.1 \times 1.1 = 1.21$, the next line $y = 1.21 \times 1.21 = 1.4641$. So the output would be 1.4641. However, when analysing the program the input value is not known. Instead, the input, `x`, is the polynomial bound $(x,0.0)$ and the output is a polynomial bound with x as an independent variable. That is, the output, `y`, is a function of the input, `x`, together with a bound on the error between this function and the original program.

Arithmetic on polynomial bounds is interpreted in the following way:

$$(P,\epsilon_1) + (Q,\epsilon_2) \rightarrow (P+Q, \epsilon_1+\epsilon_2)$$

$$(P,\epsilon_1) - (Q,\epsilon_2) \rightarrow (P-Q, \epsilon_1+\epsilon_2)$$

$$(P,\epsilon_1) \times (Q,\epsilon_2) \rightarrow (P \times Q, B(P)\epsilon_2 + B(Q)\epsilon_1 + \epsilon_1\epsilon_2)$$

$$(R,\epsilon)^{-1} \rightarrow \left(R^{-1}, \epsilon B\left(R^{-1}\right)^2\right)$$

where $B(P)$ is a constant bound on the absolute value of P calculated by summing the absolute values of its Chebyshev coefficients. Here, the rule for finding the reciprocal makes use of the inequality $B(P^2) \leq B(P)^2$.

An implementation of arithmetic on polynomial bounds may also approximate the polynomial part according to

$$(P \pm \delta, \epsilon) \rightarrow (P, \epsilon + B(\delta)) \ .$$

This allows the implementation to increase execution speeds by using approximate arithmetic on polynomials. iGen represents polynomials as a set of values at certain collocation points. We chose to use the Gauss-Lobatto collocation points because of their good convergence properties (Boyd, 2001). This is extended to the multivariate case by using a generalised, adaptive sparse grid as described in Gerstner and Griebel (2003).

So, the simple example program can be interpreted as a sequence of operations on polynomial bounds. The output can be calculated by first setting the input variable, x, equal to the polynomial bound $(x,0.0)$. The first line sets a to $(1+x,0.0)$, the second line sets y to $(1+2x+x^2,0.0)$, after the next line y becomes $(1+4x+6x^2+4x^3+x^4,0,0)$ so the output of the program can be considered to be the polynomial bound $(1+4x+6x^2+4x^3+x^4,0.0)$. If we evaluate this polynomial at $x=0.1$, for example, we get $1+0.4+0.06+0.004+0.0001=1.4641$, as we would expect.

Suppose we now tell iGen that it can apply simplifications as long as the absolute error in the output remains bounded by 0.04. The polynomial bound can be written in the Chebyshev basis as $(0.0000125T_4(x')+0.001T_3(x')+0.03005T_2(x')+0.403T_1(x')+1.03004,0.0)$ where $T_n(x')$ is the nth Chebyshev polynomial and we use the normalisation $x'=10x$ so that the independent variable lies in the range $-1\leq x'\leq 1$. This can be approximated by simply truncating the appropriate number of higher order Chebyshev terms, giving $(0.403T_1(x')+1.03004,0.0310625)$, where the bound is calculated using the inequality $|T_n(x)|\leq 1.0$. This can easily be turned back into the program

```
input(x)
  y = 4.03*x + 1.03004
output(y)
```

which approximates the original program given at the start of this section with an error bounded by ±0.0310625 and reduces the number of computational operations from 3 to 2.

2.1 Other program structures

The simple example program above illustrates the basic technique of using iGen to generate parameterisations. However, a typical programming language contains many structures that are not present in the example. In the following sections we specify how iGen deals with these structures.

2.1.1 Random numbers

As mentioned in the introduction, when we wrap the high-resolution model the transformation from low-res input to high-res input will generally make use of a random number generator. To generate random numbers we call a function, rand(), which, upon execution, returns a random floating point number in the range $(-1:1)$ with a top-hat probability distribution. When analysing a call to rand(), iGen generates a unique, especially tagged variable. The moments of each output can then be calculated by integrating over each of the tagged variables. For example, take the program

```
input(x)
  x = x + 0.005*rand()
  a = x + 1
  y = a*a
  y = y*y
output(y)
```

The first moment of y would be

$$\bar{y}=\frac{1}{2}\int_{-1}^{1}\Big[(1+4x+6x^2+4x^3+x^4)+ (0.02+0.06x+0.06x^2+0.02x^3)r_0+ (0.00015+0.0003x+0.00015x^2)r_0^2+ (5\times10^{-07}+5\times10^{-07}x)r_0^3+ 6.25\times10^{-10}r_0^4\Big]\,dr_0$$

where r_0 is the output of the random number generator. Since the outputs are always expressed in polynomial form, it is easy for iGen to integrate them symbolically. Evaluating this integral gives

$$\bar{y}=1.000050000125+4.0001x+6.00005x^2+4x^3+x^4\ .$$

2.2 Fixed loops

A loop with a fixed number of iterations can be expressed as the composition of a vector of polynomial bounds. Take, for example, the program

```
input(r)
  x = 0.0
  y = 1.0
  z = 0.0
  loop 6 times {
    dx_dt = 10.0*(y-x)
    dy_dt = r*x - y - x*z
    dz_dt = x*y - (8.0/3.0)*z
    x = x + dx_dt*0.01
    y = y + dy_dt*0.01
    z = z + dz_dt*0.01
  }
output(x)
```

which integrates the Lorenz equations over six time-steps. To deal with the loop, we first identify the variables whose initial values are used in a single iteration of the body of the loop; in this case x, y and z. We now place these into a vector of polynomial bounds

$$L=\begin{pmatrix}(x,0.0)\\(y,0.0)\\(z,0.0)\end{pmatrix}\ .$$

A single iteration of the body of the loop can now be calculated, in the usual way, as the vector

$$L = \begin{pmatrix} (x+0.1y-0.1x, 0.0) \\ (y+0.01rx-0.01y-0.01xz, 0.0) \\ (z+0.01xy-\frac{0.08}{3}z, 0.0) \end{pmatrix} .$$

The whole loop, then, is equal to the composition $L^6(x,y,z)$ and the output, x, is just the first element of this. On performing the composition and evaluating for the initial values $(0,1,0)$ for x, y and z respectively, the output equates to

$$x = -1.09964\times 10^{-15}r'^3 + 5.66995\times 10^{-7}r'^2 + 0.00169011r' + 0.455595$$

where we specify that the input, r, is in the range $0 \le r \le 28$ (which includes the value used by Lorenz) and r' is the normalised variable $r = 14(r'+1)$.

Using Chebyshev approximation (under the assumption that errors up to 10^{-4} are acceptable), this can be approximated as

$$x = 0.00171r + 0.45532 \pm 5.6\times 10^{-5} .$$

This equation converts to a computer program

```
input(r)
  x = 0.00171*r + 0.45532
output(x)
```

that calculates x in 2 arithmetic operations with an error bounded by 5.6×10^{-5}. This compares to 90 operations for the original program.

An important point to note here is that in the previous examples the calculation of the polynomial has proceeded sequentially, in much the same order as it would during an execution. The loop, L^6, however, illustrates that an analysis may proceed very differently from an execution. During an *execution* of the loop, the program pointer would loop round 6 times; during an *analysis*, on the other hand, we immediately define the meaning of the loop as L^6. This can be evaluated in any way we please. For example, we may evaluate $M = L\otimes L\otimes L$, then $L^6 = M\otimes M$, giving L^6 in 3 (albeit polynomial) operations. In some cases, there exists a closed form solution for a loop L^n in terms of n. As a simple example, suppose we have a loop with 100 iterations, and the body of the loop evaluates to $L =< 2X, Y+1 >$ for an input vector $< X, Y >$. L^n can be immediately solved as $L^n =< 2^nX, Y+n >$ giving $L^{100} =< 2^{100}X, Y+100 >$ without the need to go through the 100 iterations.

So, when a program is executed, a program pointer moves, step by step, through the program. When a program is analysed, however, it's equivalent polynomial is built up from the structures of the program. There is no program pointer, structures can be transformed in any order, the end of the program may be transformed before the beginning.

2.3 `if` statements

Consider the following program which roughly simulates a ball bouncing on the floor in a gravitational field:

```
input(z) {
  g  = 10.0
  dt = 0.01
  v  = 0.0

  loop 100 {
    z = z + v*dt - 0.5*g*dt*dt
    v = v - g*dt
    if(z < 0) {
      v = -0.8*v
      z = 0.0
    }
  }
output(z)
}
```

`z` is the height of the ball and `v` is its velocity in the upward direction. The input is the initial height that the ball is dropped from and is taken to be in the range $1 \le z \le 2$.

The new structure here is the `if` statement. This is dealt with by using the Heaviside step function, defined as

$$H(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x < 0 \end{cases}$$

where $H(0)$ is undefined.

The body of the `if` statement can be calculated in the same way as we did for fixed loops, giving the vector $P = ((0.0, 0.0), (-0.8v, 0.0))$. The condition of the `if` statement is first turned into the homogeneous form $B > 0$ (in this case we get $-z > 0$) then passed to the Heaviside step function, giving $H(-z)$ which is equal to 1 if the inequality is true, 0 otherwise. Multiplying the condition by -1 gives $H(z)$ which is the reverse: 0 if true, 1 otherwise. So the whole `if` statement is equivalent to

$$F = H(-z)P + H(z)I$$

where $I = ((z,0),(v,0))$ is the identity vector (i.e. the nth element is equal to the nth variable). So, if $z < 0$ then $F = P$ and if $z > 0$ then $F = I$. This is exactly the behaviour we require for F to be equivalent to the `if` statement[1].

So, the whole loop equates to the vector

$$L = \begin{pmatrix} (H(z+0.01v-0.0005)(z+0.01v-0.0005), 0) \\ ((1-1.8H(z+0.01v-0.0005))(v-0.1), 0) \end{pmatrix}$$

[1]Strictly speaking, the floating point variable `z` may equal 0.0 when the `if` statement is reached, in which case F would be a function of $H(0)$ which is undefined. However, this is rarely a problem in practice since we will either already be integrating over the step or we can integrate over an arbitrarily small uncertainty in the input variables, making the result formally independent of $H(0)$ as long as P remains finite.

and the whole program equates to $L^{100}((z,0),(0,0))$.

In general, the condition in an `if` statement may also contain conjunctions `&&` or disjunctions `||` so that `(A && B)` is true if and only if `A` and `B` are both true, and `(A || B)` is true if and only if `A` or `B` or both are true. A disjunction `(A || B)` is taken to be equivalent to $A+B-AB$ and a conjunction `(A && B)` is equivalent to AB.

When dealing with expressions that include the Heaviside step function, iGen first applies the following set of simple reasoning rules before the approximate expansion of the step functions into polynomial bounds.

If the argument to a step function can be proved not to cross zero then the function is replaced with 0 or 1:

$H(A) = 1$ if $B_l(A) > 0$

where $B_l(A)$ is a lower bound on A.

$H(A) = 0$ if $B_u(A) < 0$

where $B_u(A)$ is an upper bound on A. Lower and upper bounds on polynomials were calculated using $a_0 \pm B(A - a_0)$, where a_0 is the zeroth order coefficient of A.

To simplify the form of complex booleans, the following identities were used whenever the left hand sides were encountered.

$$1 - H(A) = H(-A)$$

$$H(H(A)P + H(-A)Q) = H(A)H(P) + H(-A)H(Q)$$

$$\begin{array}{ll} \text{if} & H(A)H(B)H(-C) = 0 \\ \text{and} & H(-A)H(C)H(-B) = 0 \\ \text{then} & H(A)H(B) + H(-A)H(C) = H(B)H(C)\ . \end{array}$$

The final identity is proved by noting that

$$\begin{gathered} H(A)H(B)(H(C)+H(-C)) + \\ H(-A)H(C)(H(B)+H(-B)) = \\ H(B)H(C) + H(A)H(B)H(-C) + \\ H(-A)H(C)H(-B)\ . \end{gathered}$$

This may seem like a rather arbitrary piece of reasoning, but because of the way `if` statements split the input space into two partitions, this structure was found to occur quite often. Its effect is to join together neighbouring partitions that have the same approximation.

Products of Heaviside functions of the form

$$H(P_1)H(P_2)...H(P_N)$$

can sometimes be proved to be trivially true or false, and so replaced by 1 or 0 respectively. The problem reduces to that of deciding whether a set of inequalities on polynomials is satisfiable. We used a simple algorithm based on the Gaussian elimination method. The algorithm first transforms the inequalities to equalities in the following way: Each Heaviside term $H(P_n)$ is equivalent to the inequality $P_n > 0$. Since P_n can be bounded above by $B_u(P_n)$ (as calculated using the sum of its Chebyshev coefficients) then there exists a y_n in the range $0 < y_n \le B_u(P_n)$ that satisfies $P_n - y_n = 0$. If we let

$$y_n = \frac{B(P_n)(1+z_n)}{2}$$

then z_n is in the range $[-1:1]$ and can be treated as a normal Chebyshev variable. This leads to a set of equalities

$$P'_n = P_n - \frac{B(P_n)(1+z_n)}{2} = 0$$

for all $0 < n \le N$.

Once in this form, the highest degree terms that occur in more than one equation can be successively removed by Gaussian elimination. At each stage, the bounds of the remaining polynomials are checked. If any has an upper bound that is below zero or lower bound above zero, the equation cannot be satisfied and so there is no solution. An equation P'_n was removed if it could be shown to be tautological, i.e. if it could be reduced to a form $z_n = Q$ and Q could be bounded by the interval $[-1:1]$.

We found that this algorithm detected all instances encountered in our example programs without becoming prohibitively slow. However, in the worst case, this algorithm runs in worse than exponential time so the procedure quits if the number of equalities exceeds a cutoff value. No fast algorithm for solving this problem is known, the first algorithm was due to Tarski (1951) but this also ran in worse than exponential time. Exponential time algorithms were found by Seidenberg (1954) and later by Collins (1975). More recently, a sub-exponential time algorithm has been found by Grigorev and Vorobjov (1988) but execution times remain high.

2.3.1 Conditional loops

Conditional loops can be implemented using the structures we have already described

```
while(A) {
   ...
}
```

is equivalent to

```
loop M {
  if(A) {
     ...
  }
}
```

for some M that gives the maximum number of times the `while` loop can iterate over the domain of inputs.

By insisting that M is given a finite value we are, in effect, restricting iGen's applicability to the subset of computer programs known as 'basic recursive'. These are the programs that can be proved to terminate. As it turns out (Solomonoff,

2005), almost all computer programs in practical use happen to compute basic recursive functions. Upon reflection, it is not surprising that numerical models can be shown to terminate: they are written that way. For example, if it was suspected that an algorithm could enter an infinite loop, this would in all practical respects be considered to be a bad algorithm and would be rewritten or thrown out of the model. The one exception to this is the use of randomised algorithms, some of which may technically never terminate. However, these algorithms all have the property that the probability of termination very quickly approaches 1 as the number of iterations of some loop increases. So, by limiting the number of iterations we effectively take an algorithm with a vanishingly small probability of not terminating and replace it with an algorithm with a vanishingly small probability of returning the wrong answer. So when we come to integrate over the uncertainties in the input, as long as all outputs are finite, there is always a finite M that ensures that the result is the correct answer with a vanishingly small error. We therefore restrict ourselves to the consideration of the basic recursive functions without fear that this will be a problem for our proposed application.

2.4 Arrays

Array reference and modification is performed by representing the whole array as a single polynomial with the array's index variable as an independent variable (multidimensional arrays can trivially be reduced to one dimensional arrays by using the memory address offset as the index of each element). Suppose we have an array, `A`, of size N. We choose the N equidistant points on the interval [-1:1]

$$x_n = \frac{2n}{N-1} - 1$$

where n is an integer in the range $0 \leq n < N$. From this, we let the Lagrange basis polynomials be defined as

$$l_n = \prod_{0 \leq i < N, i \neq n} \frac{x - x_i}{x_n - x_i} .$$

These polynomials have the important property that $l_n(x_m) = 0$ if $n \neq m$ and $l_n(x_n) = 1$. If we let a_i be the value of `A[i]` for all integers $0 \leq i < N$, then the $(N-1)^{th}$ degree polynomial

$$A(x) = \sum_{i=0}^{N-1} a_i l_i$$

has the property that for any integer $0 \leq j < N$, $A(\frac{2j}{N-1} - 1) = a_i$. So an array reference `A[j]` has the value

$$A\left(\frac{2j}{N-1} - 1\right) .$$

If we now define the bi-variate polynomial $L(i,x)$ as

$$L(i,x) = \sum_{j=0}^{N-1} l_j \left(\frac{2i}{N-1} - 1\right) l_j(x)$$

so that $L(i,x) = l_i(x)$ for any integer $0 \leq i < N$, then the value of an array A after an assignment operation `A[i] = Y` is equivalent to the polynomial

$$A + \left(Y - A\left(\frac{2i}{N-1} - 1\right)\right) L(i) .$$

3 Applications of iGen

3.1 Automatic derivation of the Lorenz equations

iGen was used to analyse a high-resolution model of Rayleigh-Benard convection in a 2-dimensional, laminar, incompressible fluid on an 80x28 grid. The model was wrapped so that the input was three variables (X,Y,Z). These were converted to a state of the fluid according to

$$\psi(x,z) = \frac{\sqrt{2}\kappa(1+a^2)}{a} X \sin(\pi a x)\sin(\pi z)$$

and

$$\theta(x,z) = \frac{\sqrt{2}Y\cos(\pi a x)sin(\pi z) - Z\sin(2\pi z)}{\pi R}$$

where $\psi(x,z)$ is the stream-function, $\theta(x,z)$ is the temperature field, $a = \frac{1}{\sqrt{2}}$ is the aspect ratio of the convective cells and $R = 28$ is the Rayleigh number of the flow. Similarly, the output of the high-resolution model was converted back to the (X,Y,Z) phase space by extracting the appropriate lowest modes of ψ and θ according to

$$X = \frac{1}{\sqrt{2}\kappa(1+a^2)} \int\int \psi(x,y)\sin(\pi a x)\sin(\pi z)\, dx\, dz$$

$$Y = \frac{\pi R}{\sqrt{2}a} \int\int \theta(x,y)\cos(\pi a x)\sin(\pi z)\, dx\, dz$$

$$Z = -\frac{\pi R}{2a} \int\int \theta(x,y)\sin(2\pi z)\, dx\, dz$$

The final output of the wrapped model was the average rate of change of X, Y and Z over a 0.00001s simulation.

The variables, (X,Y,Z), correspond to the variables of the Lorenz equations (Lorenz, 1963), which describe a 3-variable parameterisation of Rayleigh-Benard convection. iGen analysed the wrapped model of Rayleigh-Benard convection and produced the following simplified code:

```
input(x,y,z)
  dx_dt = 9.95076*y + 9.94443*x
  dy_dt = -0.991175*x*z - 0.999187*y
          + 27.9712*x
  dz_dt = -2.65625*z + 0.997019*x*y
output(dx_dt, dy_dt, dz_dt)
```

which is a statement of the Lorenz equations with a slight difference (less than 0.9%) in the constants. This represents an increase in execution speed of 5 orders of magnitude compared to the wrapped model. The slight difference between iGen's analysis and the Lorenz equations is attributed to the finite resolution of the grid, the finite time over which the integration was performed and the accuracy of the algorithm used to solve the Poisson equation.

3.2 Mie scattering

In order to demonstrate iGen's ability to deal with much more complex mathematical functions, a program was written to simulate the scattering of parallel light by spherical water droplets. This was done using Mie theory (Bohren and Huffman, 1998) which gives a method of solving Maxwell's equations for parallel light incident upon a sphere, using complex spherical Bessel and Hankel functions. In order to analyse this, iGen had to deal with polynomials that spanned many orders of magnitude and included sharp peaks due to resonances, without losing precision.

The program was wrapped to calculate the scattering cross section per unit mass of water for light of wavelength 500nm scattered by a thin layer of cloud. The cloud was made up of spherical water droplets with refractive index of $1.33+1\times 10^{-8}\imath$. The droplets in a cloud are not generally all of the same radius and are often assumed (Dobbie et al., 1999) to have a gamma distribution given by

$$P(r)=Ar^{\alpha}\exp^{-\beta r}$$

where

$$\alpha=\frac{1}{v_e}-3.0$$

and

$$\beta=\frac{1}{v_e r_e}$$

and A is a normalisation factor, v_e is the relative 'effective variance' of the distribution and is set to 0.172, and r_e defines an 'effective radius' of the droplets. r_e was taken to be the input of the wrapped model, and defined to lie in the range $5\mu m$ to $40\mu m$. The output of the wrapped model was defined to be the reciprocal of the scattering cross section per unit mass.

iGen was used to analyse this wrapped model and produced the simplified model for the scattering cross section K_{sca}:

$$K_{sca}=\frac{1}{660.1r_e-2.188\times 10^{-4}}$$

with an error bounded by $4m^2kg^{-1}$. This is plotted in figure 2 together with the exact result calculated using numerical integration.

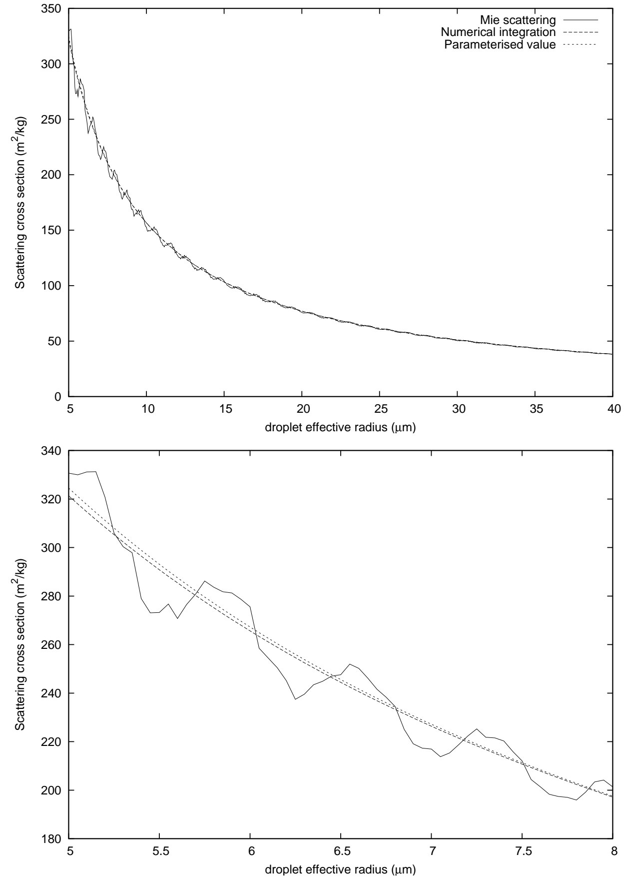

Fig. 2. Plot of scattering cross section for fixed radius droplets (solid line), numerically integrated over the droplet radius distribution (dashes), and iGen's simplified model (dots). For clarity, a magnified section is shown in the lower plot.

3.3 Entrainment in stratocumulus

In order to show that this technique scales well and can be applied to models of realistic complexity, iGen was used to create a parameterisation of entrainment in nocturnal, non-precipitating marine stratocumulus. Stratocumulus cloud occurs at the top of a well mixed, turbulent boundary layer, which underlies a much more stable free atmosphere. Radiative cooling at the top of the cloud generates turbulence which mixes or 'entrains' some of the free-atmosphere air into the boundary layer. Given the rate of this entrainment, the large scale dynamics of the boundary layer is easily calculated from budgets of mass, energy and moisture. However, no analytic derivation of this entrainment rate has been found. Lilly (1968) derives upper and lower bounds and Stevens (2002) gives details of various parameterisations. However, the simulation of marine stratocumulus remains a large source of uncertainty and error in existing climate models (Bony and Dufrence, 2005; Dufrence and Bony, 2008).

The high-resolution model in this case was a 2-

dimensional cloud resolving model with surface heat and moisture fluxes and a longwave radiation scheme (Larson et al., 2007). The cloud resolving model was wrapped so that its input was a 5-variable specification of the large scale state of the system and the output was the mean and variance per second of the entrainment velocity over the final 4 hours of a 6 hour simulation. The large scale state consisted of the variables

- Specific liquid water content at cloud top
- Jump in specific total water at inversion
- Jump in buoyancy at inversion
- Down-welling radiation just above cloud top
- Up-welling radiation just below cloud base

iGen successfully analysed the wrapped model and produced 10th degree multivariate polynomials for mean entrainment and variance of entrainment per second. These polynomials can be approximated to form a parameterisation of entrainment that executes in a few hundred arithmetic operations. Over a timestep of 30 minutes, which is typical for a climate model, the error due to approximation was shown to be small compared to the standard deviation. iGen's parameterisation of mean entrainment was shown to be in good agreement with data from the DYCOMS-II field campaign and from an intercomparison of cloud resolving models (Stevens et al., 2005). Full details of this experiment is given in Tang and Dobbie (2011).

4 Discussion

These examples provide a 'proof of concept' of the technique of automatically creating parameterisations using iGen. However, there remain many ways in which iGen could be developed further. iGen's use of an adaptive sparse grid representation for polynomials goes some way to dealing with the exponential increase in the size of polynomial approximants as the number of independent (input) variables of the parameterisation increases. However, this 'curse of dimensionality' cannot be staved off forever and iGen's performance will quickly diminish as the number of independent variables increases beyond around six, although the exact number of independent variables that can be practically analysed depends on the smoothness of the function calculated by the wrapped model. If the function contains many discontinuities or singularities then iGen's analysis will slow down and the resulting parameterisations will have wider error bounds. This could be improved by including localised adaptive grid refinement or having a piecewise polynomial representation of variables.

The error bounds reported by iGen will remain reasonably tight as long as the variables in the high-resolution model never become very sensitive to initial conditions. If they do, iGen will apply approximations in order to prevent the analysis becoming too slow. This introduces a small uncertainty to the value of the variable, and if this uncertainty is subsequently amplified in the final result, the error bounds will end up quite wide. To improve the calculation of error bounds iGen could calculate probabilistic bounds, use bounds that are functions of the input variables or use automatic differentiation in order to apply approximations more intelligently. iGen is being actively developed in this area.

5 Conclusions

In this paper we have described a new technique that allows the formal generation of fast computer models whose error is bounded compared to a high resolution model. This is important because it provides a formal, epistemic link between the results of a numerical experiment and the physical system that is being simulated. This ultimately allows conclusions about the physical system to be convincingly justified by the model results. The technique makes use of a computer program called iGen that automatically generates the source code of a parameterised model by analysing the source code of a high resolution model. iGen's ability to generate models was illustrated with a sequence of examples of increasing complexity. iGen was shown to scale up to models of realistic complexity by generating a parameterisation of entrainment in marine stratocumulus; an open problem that has been identified as a large source of uncertainty and error in existing climate models (Bony and Dufrence, 2005; Dufrence and Bony, 2008).

There is much scope for the further development of iGen and the technique described here but the authors firmly believe that iGen has the potential to become an important tool for model development.

Acknowledgements. We would like to thank Zen Internet Ltd for the loan of a computer for the stratocumulus experiment and the Natural Environment Research Council for funding this research under award number NER/S/A/2006/14148. Thanks to J.Chrimes for supplying figures from the DYCOMS-II intercomparison study and to N.Stott for his discussions on polynomials. Thanks also to Wayne and Jane at North Tea Power for much needed coffee.

References

Bohren, C.F. and Huffman, D.R.: Absorption and scattering of light by small particles. Wiley. New York. 1998.

Bony, S. and Dufrence, J.: Marine boundary layer clouds at the heart of tropical cloud feedback uncertainties in climate models. Geophysical Res. Let. 32: L20806, doi:10.1029/2005GL023851, 2005.

Boyd, J.P.: Chebyshev and Fourier spectral methods (2nd edition). Dover publications, New York. 2001.

Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. Lecture Notes in Computer Science, 33: 134-183. 1975.

Dobbie, S., J. Li, and P. Chylek: Two- and four-stream optical properties for water clouds and solar wavelengths. Journal of geophysical research. 104: 2067-2079. 1999.

Dufrence, J.L. and Bony, S.: An assessment of the primary sources of spread of global warming estimates from coupled atmosphere-ocean models. J.Clim. 21: 5135-5144, 2008.

Fahringer, T. and Scholz, B.: Advanced symbolic analysis for compilers. Lecture notes in computer Science, 2628: 1-125, 2003.

Gerstner, T. and Griebel, M.: Dimension-adaptive tensor-product quadrature. Computing, 71: 65-87, 2003.

Grigorev, D.Y. and Vorobjov, N.N.: Solving systems of polynomial inequalities in subexponential time. J. Symbolic computaion. 5: 37-64. 1988.

Khairoutdinov, M., Randall, D. and DeMott, C.: Simulations of the atmospheric general circulation using a cloud-resolving model as a superparameterisation of physical processes. J.Atmos.Sci. 62: 2136-2154, 2005.

Larson, V.E., Kotenberg, K.E. and Wood, N.B.: An analytic longwave radiation formula for liquid layer clouds. Mon.Wea.Rev. 135: 689-699, 2007.

Lilly, D.K.: Models of cloud-topped mixed layers under a strong inversion. Q.J.R.Meteorol.Soc. 94: 292-309, 1968.

Lorenz, E.N.: Deterministic nonperiodic flow. J.Atmos.Sci. 20: 130-141, 1963.

Seidenberg, A.: A new decision method for elementary algebra. Annals of Mathematics. 60: 365-374. 1954.

Solomonoff, R.: Algorithmic probability, AI and NKS. Lecture given at the Midwest NKS conference. 2005.

Stevens, B.: Entrainment in stratocumulus-topped mixed layers. Q.J.R.Meteorol.Soc. 128: 2663-2690, 2002.

Stevens, B., et.al.: Evaluation of large-eddy simulations via observations of nocturnal marine stratocumulus. Mon.Wea.Rev. 133: 1443-1462. 2005.

Tang, D.F. and Dobbie, S.: iGen: The automated generation of a parameterisation of entrainment in marine stratocumulus. To be published. 2011.

Tarski, A.: A decision method for elementary algebra and geometry. University of California Press, California. 1951.

Winsberg, E.: Simulations, models and theories: complex physical systems and their representations. Philosophy of Science. 68: S442-S454. 2001.