Acorns Software Library: User Guide

Deshana Desai, Etai Shuchatowitz, Zhongshi Jiang, Teseo Schneider, Daniele Panozzo
New York University, US

Document version 3.1 (September 2021) applicable to *Acorns* version 3.1 3.1

If you have any queries about *Acorns* that are not answered by this document then please email me at dkd266@nyu.edu.

Chapter 1

Introduction

1.1 What is Acorns?

Acorns is an algorithm that enables automatic, reliable, and efficient differentiation of common algorithms used in physical simulation and geometry processing. The algorithm is easy to integrate into existing build systems and produces dependency-free code. When used with modern compilers, our approach generates code around an order of magnitude faster than existing methods while allowing users to write their function directly in C.

The input to the algorithm is written in a subset of C99: we support arrays, for loops, nested loops, binary assignments, functions and variable declarations. The output is a set of self-contained multi-threaded C99 functions to compute the first and second order derivative.

This user guide describes how to apply the *Acorns* software library to your code, and examples of the package usage which map on to the `examples` directory of the *Acorns* software package. Section 1.2 shows how to install the package onto your system. Section 1.3 describes how to use the automatic differentiation capability of the library. Section 1.3.2 refers to the array capability and expressions supported by the library.

1.2 Installing *Acorns* on your system

The library is shipped as a conda package for ease of installation. Installing acorns from the conda-forge channel can be achieved by adding conda-forge to your channels with:

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, acorns can be installed with:

```
conda install acorns
```

Note: If you don't already have conda installed, you can install it from https://conda.io/projects/conda/en/latest/user-guide/install/index.html

1.2.1 From the source

To install the package from the source, you can perform the following steps:

```
git clone git@github.com:deshanadesai/acorns.git
cd acorns
pip install -r requirements.txt
python setup.py install
```

Note that the package requires python3 and has been tested with versions $>=$ 3.6. It also requires the pycparser and numpy libraries.

1.3 Using *Acorns* for automatic differentiation of C99 code

This section describes how to use *Acorns* to differentiate your code. *Acorns* provides the following automatic-differentiation functionality:

Full Jacobian Matrix Given a function written in C with defined output variables, differentiable input variables as well as constants; *Acorns* can compute the Jacobian matrix $\mathbf{H} = \partial\mathbf{y}/\partial\mathbf{x}$, where the element at row i and column j of $\mathbf{H}$ is $H_{i,j} = \partial y_i/\partial x_j$. This is done with a simple call to the `autodiff` function. The output jacobian matrix is returned in the form of an array called `ders` wrapped in a C function. The C function is called `compute_grad_forward` by default for forward differentiation and `compute_grad_reverse` for reverse differentiation. This output function can be called with real valued data points to compute the derivatives of the function at those points. The main function calling the compute function can be compiled with the `-O3` and optionally, `-ffast-math` flags to increase computation speed of the algorithm. *Acorns* also supports a parallel functionality for the code, the parameter `parallel` must be passed with `True` and a number of threads must be specified with the `num_threads` parameter when calling the `autodiff` function. Please refer to 1.3.3 for a more detailed example.

Full Hessian Matrix Given a function written in C with defined output variables, differentiable input variables as well as constants; *Acorns* can compute the Hessian matrix $(\mathbf{H}_y) = \partial^2\mathbf{y}/\partial\mathbf{x}^2$, where the element at row i and column j of $\mathbf{H}$ is $(\mathbf{H}_y)_{i,j} = \partial^2\mathbf{y}/\partial\mathbf{x}_i\partial\mathbf{x}_j$. The matrix is generated automatically with no manual interaction required to edit the program. To optimize over the number of calculations, we take advantage of the symmetry of the Hessian and only compute the lower triangular part of the matrix and then copy it over to the upper triangular half. The output hessian matrix is returned in the form of an array called `ders` (where `ders` is the hessian matrix stacked row-wise) wrapped in a C function. The C function is called `compute_hessian_forward` by default for forward differentiation and `compute_hessian_reverse` for reverse differentiation. Please refer to 1.3.3 for a more detailed example.

Acorns can automatically differentiate the following operators and functions:

- The standard binary mathematical operators `+`, `-`, `*` and `/`.
- The unary mathematical functions `log`, `sin`, `cos`
- The binary function `pow`

Note that at present *Acorns* is missing some functionality that you may require:

- Support for nested loops is limited to two.
- It has no support for complex numbers.
- It can be applied to C and C++ only.
- Does not currently support complicated data structures.

It is hoped that future versions will remedy these limitations.

1.3.1 Support for *For* loops

Nested for loops have limited support in the current implementation of *Acorns* . An example implementation of for loops is as follows:

```
double cross_entropy(const double **a, const double **b){
        double loss = 0;
    for(int i=0; i<2; i++){
        for(int j=0; j<2; j++ ){
                for(int k=0;k<1;k++){
            loss = loss - (b[i][j] * log(a[i][j] + 0.00001));
        }
```

```
        }
    }
    return loss;
}
```

The arrays are unrolled to produce a sequential chain of operations that are performed to get the final equation for the function to be differentiated. The program will unroll the algorithm and then parse the sequence of operations to produce a graph of all the operations. This graph will then be simplified to an AST (abstract syntax tree) which gets differentiated.

For example, the unrolled output for this program will be as follows:

```
double cross_entropy(const double **a, const double **b){
double loss = 0;
loss = (loss) - ((b[0][0]) * (log((a[0][0]) + (0.00001))));
loss = (loss) - ((b[0][1]) * (log((a[0][1]) + (0.00001))));
loss = (loss) - ((b[1][0]) * (log((a[1][0]) + (0.00001))));
loss = (loss) - ((b[1][1]) * (log((a[1][1]) + (0.00001))));
return loss;
}
```

Calling the forward differentiation method would produce the following output with the derivative:

```
void compute(double values[], int num_points, double ders[]){

        for(int i = 0; i < num_points; ++i)
        {
                double a[0][0] = values[i* 2 + 0 ];
                double a[1][1] = values[i* 2 + 1 ];
                ders[i*2+0]= (((((((0) - (((log((a[0][0] + 0.00001))) * (b[0][0]) + b[0][0] *
                ((1/((a[0][0] + 0.00001))*0)))))) - (((log((a[0][1] + 0.00001))) * (b[0][1]) +
                b[0][1] * ((1/((a[0][1] + 0.00001))*0)))))) - (((log((a[1][0] + 0.00001))) *
                (b[1][0]) + b[1][0] * ((1/((a[1][0] + 0.00001))*0)))))) - (((log((a[1][1] +
                0.00001))) * (b[1][1]) + b[1][1] * ((1/((a[1][1] + 0.00001))*0))))); //
                df/(a[0][0])
                ders[i*2+1]= (((((((0) - (((log((a[0][0] + 0.00001))) * (b[0][0]) + b[0][0] *
                ((1/((a[0][0] + 0.00001))*0)))))) - (((log((a[0][1] + 0.00001))) * (b[0][1]) +
                b[0][1] * ((1/((a[0][1] + 0.00001))*0)))))) - (((log((a[1][0] + 0.00001))) *
                (b[1][0]) + b[1][0] * ((1/((a[1][0] + 0.00001))*0)))))) - (((log((a[1][1] +
                0.00001))) * (b[1][1]) + b[1][1] * ((1/((a[1][1] + 0.00001))*0))))); //
                df/(a[1][1])
        }
}
```

1.3.2 Using *Acorns* array

Multi-dimensional arrays. *Acorns* supports array functions upto 2D arrays. Support will be extended to higher number of dimensions at a later time.

Examples of C functions as inputs with supported arrays are as follows. Example of 1D array input:

```
int function_test(double a, double p){
    double energy[0] = a*a*a*a*p+1/(p*p) - 1/p * p/a;
    return 0;
}
```

Example of 2D array input:

```
int function_test(double a, double p){
    double energy[0][0] = a*a*a*a*p+1/(p*p) - 1/p * p/a;
    return 0;
}
```

A more complex example with for loops, variable and constant initializations and matrix element assignments that can be given as input:

```
double func(const double *local_disp)
{
    const int size = 3;
    int n_grads = 4;
    double energy;

    for (long k1 = 0; k1 < size; ++k1)
    {
        for (long k2 = 0; k2 < size; ++k2)
            def_grad[k1][k2] = 0.;
    }

    for (int i = 0; i < n_grads; ++i)
    {
        for (int d = 0; d < size; ++d)
        {
            for (int c = 0; c < size; ++c)
            {
                def_grad[d][c] = def_grad[d][c]  * local_disp[i * size + d];
            }
        }
    }

    for (int d = 0; d < size; ++d)
    {
        for (int c = 0; c < size; ++c)
        {
            energy = energy + def_grad[d][c];
        }
    }
    return energy;
}
```

This code must be unrolled before differentiating it. To unroll this code saved in a file 'func.c', run:

```
acorns.unroll_file('func.c','output_func',constants=['size','n_grads','p'])
```

1.3.3 Examples

The directory `examples/sample_c_functions/` contains examples of basic usage of *Acorns* . For example, the code here shows the `pow()` functionality

```
int function_test(double r, double L){
    double energy = pow(L+r,0.5) + pow(L,2) + pow(r, -7);
    return 0;
}
```

The file is read and given as input to the `prepare_graph()` function.

```
import acorns.forward_diff as fd
with open(input_location + c_file,'r') as f:
       c_function = f.read()
ast = fd.prepare_graph(c_function)
```

This produces the abstract syntax tree from the code. If the function contained for loops, it would need to be unrolled first by calling the `unroll()` function. After this, we are ready to differentiate it using the `grad()` function call:

```
fd.grad(ast, "energy", ['L','r'], func = 'function_test',
output_filename = output_location+'/'+filename+'_grad_forward',
       output_func = 'compute_grad_forward')
```

To use reverse differentiation instead, the parameter needs to be passed the function call:

```
fd.grad(ast, "energy", ['L','r'], func = 'function_test',
reverse_diff = True, output_filename =  output_location+'/'+filename+'_grad_reverse',
       output_func = 'compute_grad_reverse')
```

Finally, to compute the hessian, the `second_der` parameter would be passed.

```
fd.grad(ast, "energy", ['L','r'], func = 'function_test',
second_der = True, output_filename = output_location+'/'+filename+'_hessian_forward',
       output_func = 'compute_hessian_forward')
```

This would produce the output C file with the function to compute the derivative as well as a header file. This can be compiled and integrated with the larger program.