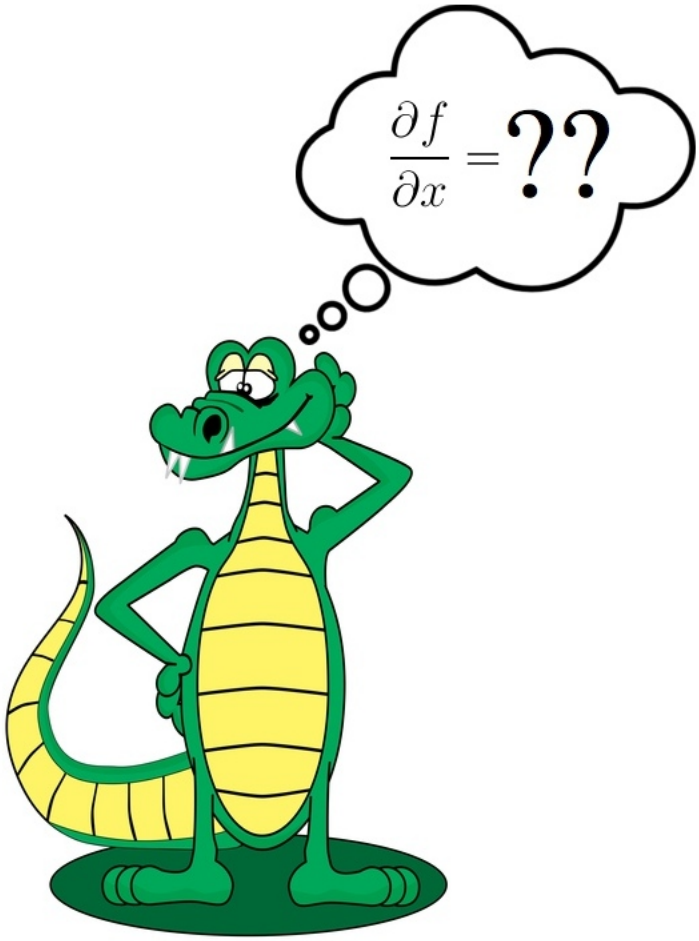

ADiGator

A Source Transformation via Operator Overloading
Toolbox for the Automatic Differentiation of
Mathematical Functions in MATLAB

Matthew J. Weinstein and Anil V. Rao

Contents

1 Introduction

ADiGator is a MATLAB automatic differentiation package which transforms user function files into derivative function files. These derivative function files are written completely in terms of the native MATLAB commands and thus may be evaluated on numeric input values to calculate the numeric derivative of the output with respect to a defined variable of differentiation. These numeric evaluations may be performed as many times as desired without generating a new derivative file, as long as the input sizes and derivative sparsity patterns are not to change. In the event that a user wishes to obtain derivatives of the same function file, but evaluated on different input sizes and/or derivative sparsity patterns, then a new derivative file must be generated. The package is particularly appealing for applications where the same derivative must be found at multiple different points, i.e. non-linear root finding/optimization, stiff ode integration, etc. For details on the methodology behind the algorithm and the overloaded class which is used, the user is referred to [1], and [2], which can be seen here and here, respectively. In the following user's guide we will explain how to use the *ADiGator* package. For more information on Automatic Differentiation in general, please refer to [3]. For an explanation on the mathematical notations used within this guide, please see the Appendix.

2 Installation

The *ADiGator* package is written entirely in MATLAB and thus should be usable on any operating system as long as MATLAB is installed. The code has been tested in MATLAB versions 7.11 through 8.2. To install the package, one needs to unpack the zip file and add two directories to the MATLAB path. This may be done as follows:

1. Unpack the zip file into a convenient location.
2. Within MATLAB, change the current directory to the location which the zip file was unpacked.
3. Run the file `startupadigator`, you can do this by typing `startupadigator` in the MATLAB command window.
4. (optional) If you wish to not do this each time you restart MATLAB, then type `savepath` in the MATLAB command window.

3 Using ADiGator

In order to obtain a numeric derivative the user must first generate a derivative file using the *ADiGator* package and may then evaluate the generated file numerically to obtain a derivative. It is stressed that after the file is generated, the software is no longer being used. Furthermore, it is noted that the derivative files are generated for a fixed input size to the user's function file. Thus, as long as the input sizes do not change, the same derivative file may be used to evaluate the derivative at multiple points. We will now explain in detail the commands required to transform a user function file into a derivative function file and then explain how to evaluate the generated derivative file. We will then demonstrate both the generation and evaluation of a derivative file using a simple example.

3.1 Generating Derivative Files

The derivative files may be generated in a three step process, where first the user must write the function file to be differentiated, second the user must define the inputs to the function file and identify which inputs contain derivative information, and then finally the transformation is initialized by using the `adigator` command. In this section we present these three steps.

3.1.1 The User Function File

In order to generate derivative files, we first need a function file of which we are to differentiate. This user written function file should be written such that one or more of the inputs is to have derivatives with respect to some *variable of differentiation* (VOD), where it is desired to determine the derivatives of the outputs of the file with respect to the same variable of differentiation. The primary restrictions placed on the input/output scheme of these functions is that 1. there is at least one input/output and 2. one of the inputs (or a field of an input structure or an element of an input cell array) has derivatives with respect to the VOD. For more user function file coding restrictions please refer to Section 5.

3.1.2 Identifying the User Function Inputs

Prior to calling the main transformation routine, `adigator`, the user must define the inputs to the function which they just created. Here we define three different types of inputs and explain how they are to be identified:

- **Derivative Inputs**: This refers to any input arrays which are to have derivatives with respect to the VOD - the user must use the `adigatorCreateDerivInput` command to generate these inputs. The syntax of the `adigatorCreateDerivInput` is as follows:

  ```
  x = adigatorCreateDerivInput(xsize,derivinfo)
  ```

  where

  - `xsize` $= [m, n]$ is the size of the input array
  - `derivinfo` gives information on the derivatives, this may be defined in one of two ways:

    1. If the input being created is the variable of differentiation (and thus has a derivative equal to the $mn \times mn$ identity matrix) then `derivinfo` can simply be set to a string name which you wish to call the VOD.
    2. Otherwise, `derivinfo` must be a structure array defining the VOD to which the input has derivatives with respect to, as well as the possible non-zero locations of the *unrolled* Jacobian of the input with respect to the VOD. These fields must be:
       * `derivinfo.vodname` = string name which uniquely defines the VOD
       * `derivinfo.vodsize` $= [p, q]$, the size of the VOD
       * `derivinfo.nzlocs` $= [\mathbf{i}, \mathbf{j}]$, where $\mathbf{i}, \mathbf{j} \in \mathbb{Z}_+^{nz}$ define the row and column locations of the non-zero elements of the $mn \times pq$ Jacobian of the input with respect to the VOD. Note: these should be in the natural MATLAB indexing order, that is, if the user has built the unrolled Jacobian in MATLAB and called it `Jac`, then `i` and `j` could be found by `[I,J] = find(Jac)`.

    Please see the Appendix for further clarification on the term "unrolled Jacobian".

- **Unknown Auxiliary Inputs**: This refers to any input arrays to the user function which do not contain derivatives with respect to the VOD, but are not a fixed, known, numeric value. That is, they may change numeric values on any given call to the user function file. - the user must use the `adigatorCreateAuxInput` command to generate these inputs. The syntax for this is as follows:

  ```
  x = adigatorCreateAuxInput(xsize)
  ```

  ,

  where `xsize` $= [m, n]$ is the size of the input array.
  NOTE: An alternative to defining Unknown Auxiliary Inputs is discussed in Section 4 by using the `auxdata` option.

- **Known Auxiliary Inputs**: This refers to any input arrays to the user function which have a fixed, known, numeric value. These inputs should simply be assigned their fixed value.

Here we note that the above three types of inputs only refer to numeric arrays, and that, if a user function was written to take a structure input with numeric fields, then the structure should be built, and the aforementioned inputs should be assigned to the proper fields. Likewise, if an input is to be a cell array, then the cell array should be built and the aforementioned input types should be assigned to the appropriate elements of the cell array.

3.1.3 Calling the Derivative File Generation

After having created the function file to be differentiated and creating all derivative/unknown auxiliary/-known auxiliary inputs, the user may now use the `adigator` command to generate the derivative file. The syntax for this call is as follows:

```
Outputs = adigator(UserFunFileName,Inputs,DerivFileName)
```

or

```
Outputs = adigator(UserFunFileName,Inputs,DerivFileName,options)
```

where

- `UserFunFileName` = string name of the user function to be differentiated
- `Inputs` = $1 \times N$ cell array, where N denotes the number of inputs to the user function and element i contains input i to the user's function file
- `DerivFileName` = string name which the derivative file is to be called
- `Outputs` = $1 \times M$ cell array, where M denotes the number of outputs of the user's function file. Cell i will contain the size and possible non-zero derivative locations of the i^{th} output.
- `options`(optional) = option structure which can be generated using the `adigatorOptions` function as defined in Section 4.

Using this command will then generate the derivative file, `DerivFileName.m`, as well as a MATLAB binary file, `DerivFileName.mat`, where `DerivFilename` is as the user specified.

3.2 Evaluating the Generated Derivative File

As previously stated, after the derivative files have been generated, the *ADiGator* software is no longer needed, but rather only the generated `.m` and `.mat` files are needed to compute the numeric derivative.[1] While the input/output structure of the generated derivative function file is similar to the input/output structure of the user defined function file, it is not exactly the same. We now look at how both the input and output structures change.

3.2.1 Derivative File Input

The only difference between the user function file input scheme and the generated derivative file input scheme is that, for any **Derivative Inputs**, these inputs must now be given as structures with a *function field* and *derivative field*. Suppose that a **Derivative Input** was defined as a $m \times n$ array with nz possible non-zero derivatives with respect to a VOD which was given the name `'vod'`. Then the input to the derivative file for this variable would have the following two fields:

- *function field*: this field will always be given the name `.f` and must be assigned the $m \times n$ numeric array corresponding to the function values.
- *derivative field*: this field will be given a name corresponding to the defined VOD name, if the VOD name was given as `'vod'`, then the field would be `.dvod`, if the VOD name was given as `'x'`, then the field would be `.dx`, etc. Furthermore, this field must be assigned a column vector of length nz corresponding to the possible non-zero derivatives of the input with respect to the VOD.

Any inputs defined as **Unknown Auxiliary Inputs** may be assigned any numeric values, as long as the array is the same size as was given to the `adigatorCreateAuxInput` command, and any inputs defined as **Known Auxiliary Inputs** should be assigned the values which were assigned prior to the call to `adigator`.

[1] The only exception to this is when differentiating a file which calls `interp2`. In this case, the file will also depend upon the ADiGator function `adigatorEvalInterp2pp`.

3.2.2 Derivative File Output

Similar to any **Derivative Inputs**, all outputs of the derivative file which correspond to numeric array outputs of the function file will now be structures with different fields. Each will contain a *function field*, `.f` containing the values of the function, and if the output contains derivatives with respect to the VOD, then it will also be assigned the following three fields:

- *derivative value field*: (for example `.dx` if the VOD name is `'x'`) This will contain a column vector of length nz corresponding to the possible non-zero elements of the output with respect to the VOD.
- *derivative size field*: (for example `.dx_size` if the VOD name is `'x'`) This will contain the size of the derivative matrix, for example, if the output variable is a vector of length m and the VOD is a vector of length n, then this will be assigned the value $[m, n]$.
- *derivative location field*: (for example `.dx_location` if the VOD name is `'x'`) This will contain a $nz \times d$ integer array of indices which map the values stored in the *derivative value field* into an array of the size stored in the *derivative size field*. Here we note that d is equal to the length of the size stored in the derivative size field, and that column i ($i = 1, \ldots, d$) of the location field gives the locations for the i^{th} dimension of the derivative array.

3.3 Illustrative Example

For this example, assume we have written some function, `y = myfun(x,k,N)`, where we wish to take the derivative of $\mathbf{y} \in \mathbb{R}^4$ with respect to the input $\mathbf{x} \in \mathbb{R}^3$, and that the input $\mathbf{k} \in \mathbb{R}^3$ is a vector of unknown auxiliary values and the input $N = 3$ is a fixed value. Prior to generating the derivative file, we would first need to identify our inputs. Supposing we wish to give our VOD the name `'x'`, we could define our **Derivative Input** in one of two ways. The first, simpler way would be:

```
x = adigatorCreateDerivInput([3 1],'x');
```

or, to the same end, we could do

```
derivinfo.vodname = 'x';
derivinfo.vodsize = [3 1];
derivinfo.nzlocs  = [1 1; 2 2; 3 3];
x = adigatorCreateDerivInput([3 1],derivinfo);
```

Next, we need to define our inputs for $\mathbf{k}$ and N, so we would do this as

```
k = adigatorCreateAuxInput([3 1]);
N = 3;
```

We now have our inputs defined and may generate the derivative code. Supposing we wished to name the derivative file `'myderiv'`, we would call `adigator` as

```
adigator('myfun',{x,k,N},'myderiv');
```

This would then generate the files `myderiv.m` and `myderiv.mat`. In order to now evaluate the derivative we need to redefine our inputs for $\mathbf{x}$ and $\mathbf{k}$. We would do this as follows:

```
x = struct('f',rand(3,1),'dx',ones(3,1);
k = rand(3,1);
```

where we are assigning random values to $\mathbf{x}$ and $\mathbf{k}$. Furthermore, we note that the derivative of $\mathbf{x}$ with respect to $\mathbf{x}$ is the 3×3 identity matrix, and that the non-zero elements are given by a vector of ones (the value assigned to `x.dx`). We could then evaluate the derivative file by

```
y = myderiv(x,k,N);
```

then `y.f` would be a vector of length 4. Supposing that the derivative of $\mathbf{y}$ has the sparsity pattern given by

$$\text{struct}(\mathbf{Jy}(\mathbf{x})) = \begin{array}{c} \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{array} \begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{bmatrix} \bullet & \bullet & \bullet \\ & \bullet & \\ & & \bullet \\ \bullet & \bullet & \end{bmatrix} \end{array} \tag{1}$$

then `y.dx` would be a column vector of length 7, `y.dx_size` would be assigned `[4, 3]` and `y.dx_location` would be `[1 1;4 1;1 2;2 2;4 2;1 3;3 3]`. Furthermore, if we wished to build a sparse MATLAB array corresponding to the Jacobian, $\mathbf{Jy}(\mathbf{x})$, we could do so using the MATLAB `sparse` command as follows:

```
Jac = sparse(y.dx_location(:,1),y.dx_location(:,2),y.dx,y.dx_size(1),y.dx_size(2));
```

If we wished to build a non-sparse MATLAB array corresponding to the Jacobian, $\mathbf{Jy}(\mathbf{x})$, we could do so using a linear index as follows:

```
Jac = zeros(y.dx_size);
index = sub2ind(y.dx_size,y.dx_location(:,1),y.dx_location(:,2));
Jac(index) = y.dx;
```

Here it is stressed that this file may be evaluated as many times as desired using different values assigned to `x.f` and `k`, which will produce different values of `y.f` and `y.dx`, but `y.dx_size` and `y.dx_location` will always be the same.

4 ADiGator Options

The *ADiGator* package currently has five different options which may given to the `adigator` command. In order to build the options structure use the following command:

```
options = adigatorOption(field1,value1,field2,value2,...)
```

A list of all options, values, and descriptions is given in Table 1.

Table 1: Options for ADiGator

Option Field	Value	Description
AUXDATA	1	**Known Auxiliary Inputs** (as defined in Section 3.1.2) will always have the same sparsity patterns as given to `adigator`, but their non-zero values may change.
	0	**Known Auxiliary Inputs** will always have the same numeric values (default)
ECHO	1	Echo to MATLAB command screen during the transformation process (default)
	0	Do not echo to MATLAB command screen
UNROLL	1	When this is set, any loops and/or sub-functions encountered will be unrolled in the generated derivative file
	0	keep loops and sub-functions rolled in the derivative file (default)
COMMENTS	1	Print comments to the derivative file giving the lines of user code which correspond to printed derivative statements (default)
	0	Do not print the comments.
OVERWRITE	1	If a `DerivFileName` is given to `adigator` such that a file already exists with the given name, setting this option will automatically overwrite the file.
	0	If this option is set, `adigator` will error out rather than overwrite the derivative file. (default)

5 User Function File Coding Restrictions

The *ADiGator* package generates derivative code by both reading the user's code and evaluating sections of the user's code on overloaded **cada** objects. This allows it to generate stand-alone derivative files, but due to the complexity of the process, certain coding restrictions are placed on the files which it can differentiate. These are mostly in place for two reasons, 1. that the package can follow the flow of the user's code and track all variables from within it, and 2. that the code has enough information to print valid derivative calculations. In this section we go over the various coding requirements which the user should follow to ensure that their function files are differentiable using *ADiGator*.

5.1 Known Numeric Values

The overloaded class which the package uses to perform the actual derivative computations is based off of having fixed, known sizes and sparsity patterns. For this reason the user's code must be written such that, given how the inputs are defined and given to the `adigator` command, all variables created within the user program must have a fixed size. As an example, consider a user function `y = myfun(x,N)` which contains the statement `y = zeros(N,1)`. If one were to identify the input `N` as an **Unknown Auxiliary Input**, then this would produce an error as this says that the variable `y` may take on *any* size. If, however, the input `N` was defined as a **Known Auxiliary Input** with value 10, then the derivative code would be generated as if the variable `N` was always 10. Similarly, all referencing/assignment indices must be known values, i.e. if the user code contains a statement `y(J) = x(I);`, then `adigator` must know the values of both `I` and `J`. Furthermore, these values must not change when evaluating the derivative file, or else the derivative calculations will be invalid. The sole exception to this rule is the use of unknown logical referencing and assignment, but this too has certain syntax requirements which are covered in Section 5.5.

5.2 Flow Control

A great deal of work has gone into the ability of the *ADiGator* package to be able to maintain the flow of the user's program within the derivative program. In this section we present the four different types of MATLAB flow control and how the *ADiGator* package deals with them.

5.2.1 Conditional `if` Statements

The user should be able to write any conditional `if` statements within their program, and any *possible* branches of the conditional block will be transcribed to the derivative program. For example, if a user function `y = myfun(x)` is given to `adigator`, the input `x` is defined as a **Derivative Input** with size [10, 1], and the function contains the following:

```
if x(1) > 1
    {calcs1}
elseif length(x) > 10
    {calcs2}
else
    {calcs3}
end
```

then `adigator` would see that `x(1)` may be any value, thus the first branch may or may not happen, that `length(x) = 10`, and thus the second branch would never happen, so the derivative code would be produced along the lines of:

```
if x(1) > 1
    {deriv calcs1}
else
    {deriv calcs3}
end
```

All outputs of conditional fragments should, however, be the same size no matter which branch is taken.

5.2.2 `for` Loop Statements

The user may write any `for` loop statements as long as the loop is set to run for a fixed, known, number of iterations. The default way of handling loops is to keep them rolled in the derivative program (that is, write out the loop), but the user may use the UNROLL option if they wish to unroll the loop. There are tradeoffs which occur when either rolling or unrolling. Namely, if the user chooses to unroll a loop which runs for many iterations, then the generated files can become extremely large as unrolling the loop will make the package print out derivative calculations for each iteration of the loop. Unrolling the loop, however, sometimes can result in a more efficient derivative file, especially if the variables within the loop change size on each iteration of the loop. Here we also note that the use of `break` and `continue` statements is allowed within loops *only* if the loops are to be rolled in the derivative file.

5.2.3 `while` Loop Statements

Currently `while` loops are not allowed in any user functions as it would be tough to see if the statements within them are differentiable. If a user wishes to differentiate a code containing a `while` loop, they are encouraged to replace the loop with a `for/if/break` sequence. That is, any `while` loop of the form:

```
count = 0;
while condition && count < 10
    count = count+1;
    {calcs}
end
```

may be replaced with a statement such as

```
for count = 1:10
    {calcs}
    if condition
        break
    end
end
```

where the *ADiGator* package can differentiate the second statement. It should be noted, however, *ADiGator* will need to run the loop for all defined iterations, so one should be careful with over exaggerating the count.

5.2.4 `switch` Statements

`Switch` statements are not currently allowed, please use `if` statements instead.

5.2.5 Short Circuit `AND/OR`

Here we note that using the short-circuit `and` (`&&`) and short-circuit `or` (`||`) are allowed, but they will be replaced with the non-short-circuit `and` (`&`) and non-short-circuit `or` (`|`) commands in the intermediate program. Thus, if the user has a line of code such as

```
if {statement1} && {statement2}
```

this will be replaced by

```
if {statement1} & {statement2}
```

thus, if evaluating `statement2` will produce an error given that `statement1` is false, then the program will not be able to be differentiated. Similarly, if the short-circuit `or` command is used such that evaluating the second statement produces an error given that the first statement is true, then the program will not be able to be differentiated.

5.3 Functions and Sub-Functions

The user's main function file given to the `adigator` command is allowed to call as many other functions/sub-functions as desired. Each called function will be opened, read, and treated as if it were a `for` loop, thus if the UNROLL option is set 1 and a sub-function is called multiple times, then multiple sub-functions will be printed to the derivative file. The restrictions on called functions and sub-functions are as follows. First, the function must be in the MATLAB path. Second, called functions and sub-functions should not share the same name, this will produce an error. Third, functions should not call themselves from within their own methods, this can create an infinite loop within the *ADiGator* algorithm and cause an error. Fourth, currently only *one* function call is allowed per line, if the user has multiple function calls on the same line then an error will be produced. Finally, each function called must have at least one input and one output. The use of the `feval` command is frowned upon here, but will work in some cases. Namely, if the function being called by the `feval` command contains no flow control and is not called from within flow control statements.

5.4 Non-Input Auxiliary Data: Global Variables and the Load Command

A user may wish to obtain auxiliary problem data using either global variables or the MATLAB `load` command. Both of these are acceptable, with some stipulations. Namely, if either is used, they should *only* be used to give auxiliary data to the user's function file, they should *not* be used to pass variables between called functions in the program being differentiated. The *ADiGator* package treats all global variables and loaded in variables as if they are **Known Numeric Inputs**, and thus all global parameters should be set to the values that they will be at the time of derivative evaluation. The last restriction is that, when using the `load` command, it must be of the syntax `var = load(.)` and not simply `load(.)`, as for the second case the algorithm would be unable to track what data was brought into the program.

5.5 Unknown Logical Referencing/Assignment

As stated in Section 5.1, the *ADiGator* algorithm is based off of having fixed, known, variable sizes. This presents an issue when dealing with unknown logical referencing and assignments, as the result of a logical reference may take on a number of different sizes depending upon the reference index. To account for this, we require that, if an unknown logical reference/assignment index is used, the same indexing variable must be used for both a reference and assignment. That is, the logical reference/assignment must be of the form:

```
ind = x > 1;
y(ind) = x(ind);
```

Furthermore, if any binary mathematical array operations (+,-,etc.) are to be performed on a variable resulting from a logical reference, then both inputs to the binary operation must be result from a logical reference of the same index. For example,

```
ind = x > 1 & x <2;
zi = x(ind).*y(ind);
z(ind) = zi;
```

is valid,

```
z(x > 1 & x <2) = x(x > 1 & x <2).*y(x > 1 & x <2)
```

is not, even though they perform the same operations.

6 Higher Order Derivatives

A nice outcome from the fact that the *ADiGator* algorithm creates stand-alone derivative code (aside from evaluation speed) is that the method is fully repeatable. Thus, the user may create n^{th} order derivative files. In order to do this, the process is repeated just as shown in Section 3, except now you are differentiating a

previously created derivative file. To illustrate, suppose we wished to take a second derivative of a function `y = myfun(x)`, with respect to the input `x`, where $\mathbf{x} \in \mathbb{R}^{10}$, and we want to call our VOD `'x'`. We would first create the first derivative file as follows:

```
x = adigatorCreateDerivInput([10 1],'x');
adigator('myfun',{x},'myderiv1');
```

If we then wished to take a second derivative with respect to $\mathbf{x}$, we would need to define a new input as the input structure to `myderiv1` is different from `myfun`. This would be done as:

```
x = struct('f',x,'dx',ones(10,1));
```

which says that the input `x.f` has the same derivatives as we defined previously, and that the input `x.dx` is a vector of ones which has no derivatives with respect to $\mathbf{x}$ (the second derivative of $\mathbf{x}$ with respect to itself is zero). We can then generate the second derivative file using the command

```
adigator('myderiv1',{x},'myderiv2');
```

In order to evaluate this function, `myderiv2`, we note that the input is exactly the same as the input to `myderiv1` since we have defined no new derivatives. So, we could evaluate it at a set of random points using the commands:

```
x.f = rand(10,1);
y = myderiv2(x);
```

Now, assuming the output has second order derivatives, the output `y` would have the following fields: `.f, .dx, .dx_size, .dx_location, .dxdx, .dxdx_size, .dxdx_location` corresponding to the function value, first derivative value, first derivative array size, first derivative mapping indices, second derivative value, second derivative array size, and second derivative mapping indices, respectively. Assuming the output `y.f` is a scalar, we could then build a sparse Gradient and Hessian using the commands:

```
Grad = sparse(ones(length(y.dx),1),y.dx_location,y.dx,1,y.dx_size);
Hes = sparse(y.dxdx_location(:,1),y.dxdx_location(:,2),y.dxdx,y.dxdx_size(1),y.dxdx_size(2));
```

This process may be repeated as many times as desired.

Here we note that the *ADiGator* algorithm is cognizant of when it is differentiating a function which it created, thus a bunch of embedded structures are not required for the inputs/outputs of the higher order derivative files. Furthermore, it knows what derivatives were taken on previous transformations and identifies these using the string name given to the variable of differentiation. This allows the user to take derivatives with respect to different variables if it is desired, but if the same VOD name is used as a previous transformation, then the given **Derivative Inputs** should reflect those given in the previous transformation.

Here we also note that the user can write functions which call previously created derivative files and then differentiate the new file and that the algorithm will still recognize the previously created derivative files as their own and differentiate accordingly.

7 Differentiating Vectorized Code

The use of the vectorized differentiation is for problems of the form $\mathbf{f}(\mathbf{x}(s))$, where s is a scalar, independent variable. It is easiest to think of s as being a representation of time such that $\mathbf{f}$ at $s = t$ is only a function of $\mathbf{x}$ at $s = t$. If this is the case, and a user's code is written such that it computes the values of $\mathbf{f}$ at a set of time points, given the inputs of $\mathbf{x}$ at a set of time points, then the code may be differentiated in a vectorized manner. More specifically, if the code is written to compute $\mathbf{F}(\mathbf{X}(\mathbf{s})) : \mathbb{R}^{N\times n} \to \mathbb{R}^{N\times m}$, where

$$\mathbf{X}(\mathbf{s}) = \begin{bmatrix} x_1(s_1) & \cdots & x_m(s_1) \\ x_1(s_2) & \cdots & x_m(s_2) \\ \vdots & \ddots & \vdots \\ x_1(s_N) & \cdots & x_m(s_N) \end{bmatrix}, \quad \mathbf{F}(\mathbf{X}(\mathbf{s})) = \begin{bmatrix} f_1(\mathbf{x}(s_1)) & \cdots & f_n(\mathbf{x}(s_1)) \\ f_1(\mathbf{x}(s_2)) & \cdots & f_n(\mathbf{x}((s_2)) \\ \vdots & \ddots & \vdots \\ f_1(\mathbf{x}(s_N)) & \cdots & f_n(\mathbf{x}((s_N)) \end{bmatrix}, \tag{2}$$

where it is important that the code be written such that N may be *any* positive integer. When this is the case, we compute the Jacobian $\mathbf{Jf}(\mathbf{x}(s))$, but do so such that the non-zero elements of $\mathbf{Jf}(\mathbf{x}(s))$ are computed for N values of s, where, again, N may take on any integer value.

7.1 Illustrative Example of Vectorized Differentiation

In order to differentiate in the vectorized mode the user must simply identify their vectorized inputs. To do this, the vectorized dimension is just given the value `Inf`. So, if the user has a function, `Y = myfun(X)`, of the above form with $n = 3$, $m = 4$, then to generate the **Derivative Input**, the following command would be used:

```
X = adigatorCreateDerivInput([Inf 3],'X');
```

To then create a vectorized derivative file, `myderiv`, we would call

```
adigator('myfun',{X},'myderiv');
```

In order to now evaluate the file `myderiv` we must define the function field of `X.f` and the derivative field `X.dX`. Supposing we wish to evaluate at a set of random function inputs at $N = 10$ values of s, we would define the function field as

```
x.f = rand(10,3);
```

Now, here we note that the derivative field needs to be of the dimension $N \times nz$, where nz corresponds to the number of non-zeros of the input Jacobian (in this case $\mathbf{Jx}((\mathbf{x}(s)))$, and N corresponds to the vectorized dimension. In this case, $nz = 3$, and

$$\frac{\partial x_i(s)}{\partial x_j(s)} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \qquad \forall s \tag{3}$$

thus the derivative field should be defined as

```
x.dX = ones(10,3);
```

We may then evaluate the derivative function using the command

```
y = myderiv(x);
```

Now, the output fields of $\mathbf{y}$ will have the same form as in the non-vectorized case, with a few exceptions. First, the values assigned to `y.dX_size` and `y.dX_location` are the size and non-zero locations of the Jacobian $\mathbf{Jy}(\mathbf{x}(s))$ evaluated at a single time point. Next, the derivative values stored in `y.dX` are stored in an array of size $N \times nz$, where nz is the number of possible non-zero derivatives in the Jacobian $\mathbf{Jy}(\mathbf{x}(s))$ evaluated at a single point. So, to build the Jacobian $\mathbf{Jy}(\mathbf{x}(s))$ evaluated at $s = s_1$, we could do so as

```
Jac1 = sparse(y.dX_location(:,1),y.dX_location(:,2),y.dX(1,:),y.dX_size(1),y.dX_size(2));
```

7.2 Projecting Vectorized Derivatives into an Unrolled Jacobian

Now, if we let $x_i(\mathbf{s})$ $(i = 1, \ldots, n)$ and $f_j(\mathbf{X}(\mathbf{s}))$ $(j = 1, \ldots, m)$ be column vectors of length N, where the k^{th} elements are $x_i(s_k)$ and $f_j(\mathbf{x}(s_k))$ $(k = 1, \ldots, N)$, respectively, we can then define

$$\mathbf{x}^\dagger(\mathbf{s}) = \begin{bmatrix} x_1(\mathbf{s}) \\ x_2(\mathbf{s}) \\ \vdots \\ x_n(\mathbf{s}) \end{bmatrix}, \quad \mathbf{f}^\dagger(\mathbf{x}^\dagger(\mathbf{s})) = \begin{bmatrix} f_1(\mathbf{X}(\mathbf{s})) \\ f_2(\mathbf{X}(\mathbf{s})) \\ \vdots \\ f_m(\mathbf{X}((\mathbf{s})) \end{bmatrix}. \tag{4}$$

Now, supposing the user wished to build the unrolled Jacobian $\mathbf{Jf}^\dagger(\mathbf{x}^\dagger(\mathbf{s}))$, we have supplied the function `adigatorProjectVectLocs`, which takes the output locations of the non-zeros of the Jacobian $\mathbf{Jf}(\mathbf{x}(s))$, together with the length of the vectorized dimension, N, to build the non-zero locations of the unrolled Jacobian $\mathbf{Jf}^\dagger(\mathbf{x}^\dagger(\mathbf{s}))$. For the above example, we could build the unrolled Jacobian follows:

```
[I,J] = adigatorProjectVectLocs(10,y.dX_location(:,1),y.dX_location(:,2));
Jacdag = sparse(I,J,y.dX,y.dX_size(1)*10,y.dX_size(2)*10);
```

Here we note that one should be careful when using this command as it will only be correct if the input's first dimension is vectorized and the output's first dimension is vectorized. That is, $\mathbf{x}^\dagger(\mathbf{s})$ would correspond to `x.f(:)` and $\mathbf{f}^\dagger(\mathbf{x}^\dagger(\mathbf{s}))$ would correspond to `y.f(:)` in our example, due to how we have arranged the data.

7.3 Vectorized Coding Restrictions

The first vectorized coding restriction is that, if the vectorized dimension of the input is N, then all vectorized variables must be of size $N \times n$, where n must be a known value (and not equal to N). Furthermore, one may not sum over any vectorized dimension, as this results in derivatives of the output being dependent upon all points in time. Similarly, referencing off of the vectorized dimension, but not taking the entire row/column is not allowed, e.g. if `X` is of size $N \times 3$, you cannot perform `xi = X(1,:)`, but you can perform `xi = X(:,1)`. Finally, you are not allowed to loop on a vectorized dimension, e.g. if `Y` is of size $1 \times N$, you may not do `for yi = 1:Y`. It is noted here that unknown logical referencing/assignments are coded up for vectorized dimensions and that these may be used (in accordance with the guidelines given in Section 5.5) instead of a `for` loop if you need to search through the vectorized dimension.

8 Generating Files for GPOPS II

GPOPS II is a commercial MATLAB optimal control software which utilizes direct collocation methods to solve optimal control problems. The creators of *ADiGator* have collaborated with those of *GPOPS II* in order to allow for *ADiGator* to supply first- and second-order derivatives to *GPOPS II*. In order to generate the derivative files using *ADiGator* please see the file `adigatorGenFiles4gpops2`. This function utilizes the vectorized and non-vectorized modes in order to generate the derivative files required by *GPOPS II* given the `setup` structure which is used by the `gpops2` function. Like all *ADiGator* generated files, these generated files will not need to be changed unless the user changes their continuous or endpoint functions.

9 Debugging

During the transformation from user code to derivative code, the *ADiGator* algorithm never actually evaluates the user's code, but rather copies various parts of it, writes them to another file, and then evaluates that file. This can sometimes make debugging difficult, particularly because the MATLAB error messages do not point to the user's file with the errors. Rather, they will usually point to a user's line of code which was copied to another file. We are working on a way to point to the user's file, but for now the user will need to deal with the inconvenience. In this section we will go over some of the common errors that may occur when using the *ADiGator* software.

9.1 Error in User Code or Inputs to ADiGator

Prior to performing any transformation, the algorithm first attempts to evaluate the user's function numerically on what it believes to be the inputs. If you receive the error

```
Error in initial test evaluation of user file on given input info
```

then one of two things has occurred: either there is an error in the user's file or the inputs to the user's file have not been properly given to `adigator`. To debug this, you should first ensure that your function evaluates on numeric inputs without errors. If you are still receiving the error, then refer to Section 3.1.2 for the proper input scheme.

9.2 Non-Overloaded Functions

The algorithm evaluates the user's statements on an overloaded **cada** class, so, if the user's code contains calls to functions which are not overloaded an error will be produced as

```
Undefined function 'somefunction' for input arguments of type 'cada'.
```

In this case, the only way to use that function within the user's code is to overload the function. If the calculation in question may be performed prior to the user's code (i.e. you do not need to take derivatives of the function), then this is a simple solution. If however, the function in question is operating on objects which have derivatives, then the user may either contact the author in order to see if the function can be

overloaded, or rewrite their code to use a different function. Here we note that, in order to overload an operation, there *must* be a derivative rule defined for said operation, thus operations such as `min` and `max` may not be overloaded as they do not have a defined derivative rule. A list of all the currently overloaded operations may be seen in the `adigator/@cada/` directory.

9.3 Errors Regarding "Strictly Symbolic" Inputs

As stated in Section 5.1, in order to create an array, the size of the array must be a known numeric value. Similarly, in general, when performing a reference/assignment, the reference/assignment index must be a known numeric value. When these types of operations are performed and the algorithm does not know the value of the size and/or index, then an error will be produced. For example, if one were to perform the operation `y = ones(N,1)`, and the variable `N` does not have a known numeric value, then the error

```
Cannot pre-allocate a stricly symbolic size
```

would be produced. Similarly, if one were to perform the operation `y = x(I)` and the variable `I` does not have a known numeric value, then the error

```
Cannot perform strictly symbolic referencing/assignment
```

would be produced. In order to fix this, ensure that, if these sizes/indices are inputs, that they are given such that they are **Known Auxiliary Inputs** as shown in Section 3.1.2. If they are values calculated from within the file, make sure that they are created by operating on known numeric objects. Also, note that the sizing operations `size,numel,length` will always produce a known numeric value.

9.4 Errors in the Derivative and/or Derivative Code

If you receive an error in the evaluation of the derivative code (on proper numerical inputs), then you have likely found a bug in the algorithm. Please report this. Similarly, if a derivative is computed incorrectly, then you have more than likely found a bug in the algorithm. Please report this as well. If an element of the derivative is being computed as `NaN`, then the derivative is likely undefined at the evaluation point. A common occurrence of this is evaluating the derivative of the `sqrt` function at 0 when the derivative of the `sqrt` argument is also zero. That is, if `y = sqrt(x)`, the derivative rule is produced along the lines of `dy = (1/2)/sqrt(x)*dx`; if this is then evaluated when `x=0` and `dx=0`, then `1/sqrt(x)` goes to infinity, and zero times infinity is undefined. Thus, the derivative code isn't wrong, the derivative is just undefined at that point.

10 List of Included Examples

In this section we present an explanation of the examples included with the *ADiGator* package.

10.1 Basic First Derivatives

In this section we present two examples which take the first derivative of vector functions of vectors and compare against MATLAB's `numjac` finite differencing tool.

10.1.1 Arrowhead

This is an example taken from Section 7.4 of [3]. Here we take the derivative of a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, where

$$f_1(\mathbf{x}) = 2x_x^2 + \sum_{i=1}^{n} x_i^2, \quad f_j(\mathbf{x}) = x_1^2 + x_j^2, \quad (j = 2, \ldots, n) \tag{5}$$

This is coded up such that the Jacobian, $\mathbf{Jf}(\mathbf{x})$ is built using *ADiGator*, finite differencing, and compressed finite differencing. Here the user is free to change the the parameter `N` within the `main.m` code in order to see that the *ADiGator* algorithm becomes more useful as the problem size increases. Furthermore, you can see that there is a certain amount of overhead inherent with generating the derivative code, but, if you wish to evaluate the code many times, this overhead becomes less of an issue.

10.1.2 Polynomial Data Fitting

This example was taken from [4] which is to determine the coefficients of the m-degree polynomial $p(x) = p_1 + p_2 x + p_3 x^2 + \cdots + p_m x^{m-1}$ that best fits the points $(x_i, d_i),\ (i = 1, \ldots, n),\ (n > m)$, in the least squares sense. This polynomial data fitting problem leads to an overdetermined linear system $\mathbf{Vp} = \mathbf{d}$, where $\mathbf{V}$ is the Vandermonde matrix,

$$\mathbf{V} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{m-1} \\ \vdots & \vdots & & \vdots & \\ 1 & x_n & x_n^2 & \cdots & x_n^{m-1} \end{bmatrix}. \tag{6}$$

As with the previous example, the user should note that the overhead associated with generating the derivative file becomes less as the problem size increases, as well as the number of required derivative evaluations. Unlike the previous example, it is noted that the resulting Jacobian, $\mathbf{Jp}(\mathbf{x})$, is full (i.e. has no known non-zero elements), thus, after the derivative file is evaluated, the Jacobian is simply built using the reshape command:

```
dpdx = reshape(p.dx,p.dx_size);
```

10.2 Stiff Ordinary Differential Equations

In this section we supply derivatives to the MATLAB ODE integrator, `ode15s`, to integrate two different ODEs.

10.2.1 Brusselator

In this example, we integrate the well known Brusselator system of [5]. The dynamics of the system are given as

$$\begin{aligned} \dot{u}_i &= 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ \dot{v}_i &= 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{aligned} \qquad (i = 1, \ldots, N) \tag{7}$$

with initial conditions

$$\begin{aligned} u_i(0) &= 1 + \sin\left(2\pi \tfrac{i}{N+1}\right) \\ v_i(0) &= 3 \end{aligned} \qquad (i = 1, \ldots, N) \tag{8}$$

The code contained in `adigator/examples/stiffodes/brusselator/main.m` will generate the derivative file for the system using *ADiGator*. It then integrates the system with `ode15s` three different times. The first time it does not supply derivatives, the second time it supplies the sparsity pattern (as calculated from the outputs of the `adigator` transformation command), and finally it supplies derivatives using the *ADiGator* generated file. Here we note that the *ADiGator* generated derivative file may not be given directly to `ode15s` as the input/output scheme of the generated files is different from the required input/output scheme for use with `ode15s`. Thus, a wrapper file is written such that the wrapper file has the input/output scheme required of `ode15s`. The wrapper file then uses its inputs to call the generated derivative file, and then build the Jacobian output using the outputs of the derivative file. The user is free to change the dimension N and the time span over which to integrate. They should notice that supplying no derivative information is extremely slow, while supplying just the sparsity pattern or the sparsity pattern plus derivatives yield similar solve times.

10.2.2 DCAL Control of Two-Link Robot Manipulator

This example is a simulation of the experiment presented in [6]. The paper derives a control for the robotic model:

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{V}_m(\mathbf{q}, \dot{\mathbf{q}})\mathbf{q} + G(\mathbf{q}) + \mathbf{F}_d(\dot{\mathbf{q}}), \tag{9}$$

where $\mathbf{q}(t)$ is a position vector. The control, $\boldsymbol{\tau}$, is computed as a function of $\mathbf{q}$, $\mathbf{Y}_d$ and $\dot{\mathbf{Y}}_d$, where

$$\mathbf{Y}_d = \mathbf{Y}_d(\mathbf{q}_d, \dot{\mathbf{q}}_d, \ddot{\mathbf{q}}_d) \tag{10}$$

and the vector $\mathbf{q}_d(t)$ is the desired trajectory. For this simulation, we have a two-link robot, thus $\mathbf{q}, \mathbf{q}_d, \boldsymbol{\tau} \in \mathbb{R}^2$, with the desired trajectory given as

$$q_{d1}(t) = q_{d2}(t) = 0.7\sin(2t)\left(1 - e^{-.3t^3}\right). \tag{11}$$

Prior to integrating the differential equation is is required that the time derivatives $\dot{\mathbf{q}}_d$, $\ddot{\mathbf{q}}_d$, and $\dot{\mathbf{Y}}_d$ are derived. Rather than doing this by hand, this is done using *ADiGator* as follows. First, we differentiate the desired trajectory function `qd = getqd(t)` with respect to `t`. We then differentiate the resulting derivative file to get a file for $\ddot{\mathbf{q}}_d$, and then once more differentiate the resulting derivative file to get a file which calculates $\dddot{\mathbf{q}}_d$. We then define the vector $\mathbf{Q} = [\mathbf{q}_d^T, \dot{\mathbf{q}}_d{}^T, \ddot{\mathbf{q}}_d{}^T]^T$, where $\mathbf{Y}_d = \mathbf{Y}_d(\mathbf{Q})$. The calculations to get $\mathbf{Y}_d$ are given in the file `Yd = getYd(Q)`. We then take the derivative of the file `getYd` with respect to time in order to get a file which computes $\dot{\mathbf{Y}}_d(\mathbf{Q}, \dot{\mathbf{Q}})$. The control laws written in the file `getDCALcontrol` are then written such that they call these generated derivative files.

Having generated these time derivative files, we then solve the ODE using `ode15s`. Here we note that we are just integrating the robot positions and velocities, but also an internal filter variable and a variable used in the parameter update law. Thus our states which we are integrating are defined as $\mathbf{x} = [\mathbf{q}^T, \dot{\mathbf{q}}^T, \mathbf{p}^T, \mathbf{z}^T]^T$, where $\mathbf{q} \in \mathbb{R}^2$ is the robot link position, $\dot{\mathbf{q}} \in \mathbb{R}^2$ is the robot link velocity, $\mathbf{p} \in \mathbb{R}^2$ is the internal filter variable, and $\mathbf{z} \in \mathbb{R}^5$ is the variable used in the parameter update law. The dynamic equations associated with all variables is calculated from within the `xdot = TwoLinkSys(t,x)` file (which calls `getDCAlcontrol`). We then take the derivative of the file `TwoLinkSys` with respect to `x` and supply this to the ODE solver `ode15s` in order to integrate the system.

As with the previous example, a wrapper is used since the input/output scheme required by `ode15s` is different from that of *ADiGator* generated files. Within the `main.m` file (which runs the simulation), the user is free to change the `supplyderiv` flag (to supply derivatives or not), the `displayplots` flag (to display plots or not), the `probinfo.noiseflag` (to add noise or not), or the value of final time, `tf`. Increasing the final time, not supplying derivatives, or adding noise will all increase the simulation run time.

10.3 Constrained Optimization

In this section we supply derivatives to the MATLAB constrained optimization solver `fmincon` to solve three different problems. The first example is given as a simple demonstration on how to supply Objective Gradients, Constraint Jacobians, and Lagrangian Hessians using *ADiGator*. In the second example we divide our problem into a vectorized portion and a non-vectorized portion. Finally, in the last example, we take full advantage of the vectorized nature of the problem.

10.3.1 Simple Example

The code for this example may be found in `adigator/examples/optimization/simple_fmincon`. For this example, we use the problem taken from the MATLAB optimization toolbox help documentation. The problem is as follows:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^2} f(\mathbf{x}) &= e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \\ &\text{such that} \\ \mathbf{c}(\mathbf{x}) = \begin{matrix} x_1x_2 - x_1 - x_2 + 1.5 \\ -x_1x_2 - 10 \end{matrix} &\leq \mathbf{0} \end{aligned} \quad . \tag{12}$$

We use this somewhat simple problem to demonstrate how to create an *objective Gradient*, *constraint Jacobian*, and *Lagrangian Hessian*. Furthermore, we show how these may be used with MATLAB's non-linear programming solver, `fmincon`. If the user does not have the optimization package, then `fmincon` will not be called, but they may still see how to generate the mentioned derivatives. We do this as follows. First, we solve the problem without supplying derivatives using the active-set method of `fmincon`. Next, we generate objective Gradient and constraint Jacobian files and supply these to `fmincon` to solve the problem again with the active-set method. We then move onto generating Lagrangian Hessians and supplying them to `fmincon`. This is done in two ways. The first is to write a file which computes the Lagrangian, and differentiate it twice using *ADiGator*. In the second way, we take advantage of the fact that we previously created derivative files to compute the objective Gradient and constraint Jacobian, thus we write a file which computes the

Lagrangian Gradient by calling these files. The Lagrangian Gradient file is then differentiated, achieving the same thing as if we were to simply differentiate the Lagrangian twice. Here we stress that various wrapper files must be made in order to use the *ADiGator* generated files with `fmincon` as the input/output scheme required by `fmincon` is different from the input/output scheme of *ADiGator* generated files. These files are included, but are not automatically generated.

10.3.2 Brachistochrone

In this example, we solve a discretized form of the continuous time optimal control problem:

$$\min_{\mathbf{x}(t),u(t),t_f} t_f \tag{13}$$

subject to the dynamic constraints

$$\mathbf{f}(\mathbf{x}(t),\mathbf{u}(t)) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \end{bmatrix} = \begin{bmatrix} x_3(t)\sin u(t) \\ -x_3(t)\cos u(t) \\ g\cos u(t) \end{bmatrix} \tag{14}$$

and boundary conditions

$$\begin{array}{llll} x_1(0) = & 0, & x_1(t_f) = & 2 \\ x_2(0) = & 0, & x_2(t_f) = & -2 \\ x_3(0) = & 0. & & \end{array} \tag{15}$$

To solve this problem we use a Hermite-Simpson Separated collocation method. We do this using K equally spaced intervals with a discrete point at the beginning and middle of each interval, as well as at a final point. This results in $N = 2K+1$ discrete time points, t_j $(j = 1,\ldots,N)$. Here we introduce our discretized state, $\mathbf{X} \in \mathbb{R}^{N\times 3}$ where

$$X_{i,j} = x_i(t_j) \quad (i = 1,2,3) \quad (j = 1,\ldots,N) \tag{16}$$

and our discretized control, $\mathbf{U} \in \mathbb{R}^N$, where

$$U_j = u(t_j) \quad (j = 1,\ldots,N). \tag{17}$$

We also introduce the matrix function $\mathbf{F}(\mathbf{X},\mathbf{U}) \in \mathbb{R}^{N\times 3}$, where,

$$F_{i,j} = f_i(\mathbf{X}_j^T, U_j), \quad (i = 1,2,3) \quad (j = 1,\ldots,N), \tag{18}$$

where $\mathbf{X}_j$ is the j^{th} row of the matrix $\mathbf{X}$. We now give the discretized problem as:

$$\min_{\mathbf{X},\mathbf{U},t_N} t_N \tag{19}$$

subject to the equality constraints

$$\begin{array}{c} X_{i,2k+1} - \frac{1}{2}(X_{i,2k+2} + X_{i,2k}) - \frac{t_N}{8K}(F_{i,2k} - F_{i,2k+2}) = 0 \\ X_{i,2k+2} - X_{i,2k} - \frac{t_N}{6K}(F_{i,2k+2} + 4F_{i,2k+2} + F_{i,2k}) = 0 \\ (i = 1,2,3) \quad (k = 1,\ldots,K) \end{array} \tag{20}$$

and simple bounds

$$\begin{array}{c} \mathbf{X}_{min} \le \mathbf{X} \le \mathbf{X}_{max} \\ \mathbf{U}_{min} \le \mathbf{U} \le \mathbf{U}_{max} \\ t_{Nmin} \le t_N \le t_{Nmax}, \end{array} \tag{21}$$

where we are using the notation that $\mathbf{X}_i$ is the i^{th} row of the matrix $\mathbf{X}$, and that the boundary conditions of Equation 15 are included in the simple bounds of Equation 21. There are five different driver files which will solve this problem in the `adigator/examples/optimization/brachistochrone` directory. The file `main_noderivs` solves it without supplying derivative information. The files `main_basic_1stderivs` and `main_basic_2ndderivs` solve the problem by supplying first and second derivative information, respectively, without taking advantage of the vectorized nature of $\mathbf{F}(\mathbf{X},\mathbf{U})$. And the files `main_vect_1stderivs`

and `main_vect_2ndderivs` solve the problem by supplying first and second derivatives, respectively, while taking advantage of the vectorized nature of $\mathbf{F}(\mathbf{X}, \mathbf{U})$. Here we note that we do not use *ADiGator* to compute the Objective Gradient as this is a simple calculation that does not require automatic differentiation. Furthermore, all of these files solve the problem four different times using values of $K = 5$, $K = 10$, $K = 20$, and $K = 40$, where, for the last three iterations the solution of the previous iteration is used as an initial guess to the current iteration.

Supplying Derivatives without Vectorization

In order to supply a Constraint Jacobian in the most straightforward way, we write the function `[C,Ceq] = basic_cons(z,probinfo)`, where the input vector is $\mathbf{z} = [\mathbf{X}^{\dagger T}, \mathbf{U}^T, t_N]^T$, where $\mathbf{X}^\dagger$ is the unrolled form of $\mathbf{X}$ (as described in the Appendix). The file `basic_cons` calls the file `F = dynamics(X,U)` and uses the output to build the constraints of Equation 20. So, in order to get the Constraint Jacobian, we simply apply *ADiGator* to the `basic_cons` file using `z` as our variable of differentiation. To then supply the Constraint Jacobian to `fmincon` using `fmincon`'s desired input/output scheme, we write the wrapper `basic_conswrap` which calls our generated derivative file and builds the Constraint Jacobian.

In order to supply the Lagrangian Hessian, we first write a file, `GL = basic_laggrad(z,lambda,probinfo)`, which uses our constraint derivative file to build the Lagrangian Gradient:

$$\bigtriangledown_{\mathbf{z}} L = \bigtriangledown_{\mathbf{z}} t_f + \boldsymbol{\lambda}^T \bigtriangledown_{\mathbf{z}} \mathbf{c}(\mathbf{z}), \tag{22}$$

where $\bigtriangledown_{\mathbf{z}} L$, $\bigtriangledown_{\mathbf{z}} t_f$, $\boldsymbol{\lambda}$, and $\bigtriangledown_{\mathbf{z}} \mathbf{c}(\mathbf{z})$ represent the Lagrangian Gradient, Objective Gradient, Lagrange multipliers, and Constraint Jacobian, respectively. We then apply *ADiGator* to the file `basic_laggrad` using `z` as our variable of differentiation to create a file which is used to build the Lagrangian Hessian. In order to supply the Lagrangian Hessian to `fmincon`, we again write a wrapper which will call the generated derivative file `basic_laggrad_z` and build the Lagrangian Hessian using the required input/output scheme of `fmincon`. Here we note that, since we solve the problem at four different values of K, that this changes our problem size, and thus all derivative files must be generated prior to each call to `fmincon`.

Supplying Derivatives with Vectorization

Here we note that a part of our problem, namely, $\mathbf{F}(\mathbf{X}, \mathbf{U})$, is of the vectorized nature in that $\mathbf{F}_i$ is only a function of $\mathbf{X}_i$ and $\mathbf{U}_i$ $(i = 1, \ldots, N)$. As such, we may use *ADiGator* in the vectorized mode in order to differentiate this sub-problem. In order to do so, we first define a variable $\mathbf{y}(t) = [\mathbf{x}^T(t), u(t)]^T$, we then apply *ADiGator* in the vectorized mode to the file `F = dynamics(X,U)` with $\mathbf{y}(t)$ as our variable of differentiation. This essentially gives us a file which will compute $\mathbf{Jf}(\mathbf{y}(t))$ at the time points $t_1, \ldots, t_N$, given the inputs $\mathbf{X}$ and $\mathbf{U}$, where N may be any positive integer. In order to supply the Constraint Jacobian, we then write a file `Ceq = vect_cons(X,F,tf,probinfo)`, where, unlike in the non-vectorized case, this takes `F` as an input rather than calling the dynamics file. We then use the sparsity pattern of $\mathbf{Jf}(\mathbf{y}(t))$ together with the function `adigatorProjectVectLocs` to define the sparsity pattern of $\mathbf{JF}(\mathbf{z})$, given a fixed value of N, and apply *ADiGator* in the *non-vectorized* mode to the file `vect_cons` with `z` as our variable of differentiation. In order to supply the Constraint Jacobian to `fmincon`, we then write a wrapper file `vect_conswrap` which calls our vectorized dynamics derivative file, and uses the derivative outputs as inputs to the constraint derivative file and builds the Constraint Jacobian.

In order to supply the Lagrangian Hessian in this manner, we first must create a file which computes the second derivatives of the dynamics file, that is, $\bigtriangledown^2_{\mathbf{y}(t)} \mathbf{f}(t)$. This is done by applying *ADiGator* to our previously created vectorized dynamics derivative file, `dynamics_Y(X,U)`, again using $\mathbf{y}(t)$ as our variable of differentiation. Now, in order to get the Lagrangian Hessian, we first write a file which computes the Lagrangian Gradient, and take the derivative with respect to `z`. In this example, we wrote this file as `GL = vect_laggrad(X,dX,F,dF,tf,dtf,probinfo)`, where `X`,`F`, `dF`, and `tf` are all identified as derivative inputs. Furthermore, the input `dF` is the non-zero derivatives of $\mathbf{JF}(\mathbf{z})$, and as such, identifying the non-zero locations of the derivative of `dF` with respect to `z` can be tricky. After identifying the derivative inputs properly, *ADiGator* is called in the non-vectorized mode on the file `vect_laggrad` using `z` as the variable of differentiation. In order to supply the Lagrangian Hessian we write the file `vect_laggradwrap` which first calls the second derivative of the dynamics file and then calls the derivative of the Lagrangian Gradient

file. Here we note that since the derivatives of the dynamics file are done in the vectorized mode, that they only need to be generated once. Since the other derivative files are generated in the non-vectorized mode, they must be generated each time the value of K is changed. We also note that, for this example, one does not gain a lot of efficiency by taking advantage of the vectorized nature of the dynamics (primarily because the dynamics are fairly simple), but hopefully it will give the user some insight on how a problem may be separated into a vectorized part and a non-vectorized part.

10.3.3 Minimum Time to Climb of Hypersonic Aircraft

In this example, we solve a discretized form of the continuous time optimal control problem:

$$\min_{\mathbf{x}(t),u(t),t_f} t_f \tag{23}$$

subject to the dynamic constraints

$$\mathbf{f}(\mathbf{x}(t),\mathbf{u}(t)) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \end{bmatrix} = \begin{bmatrix} x_2(t)\sin x_3(t) \\ \frac{\zeta(\mathbf{x}(t))-\theta(\mathbf{x}(t))}{c_2} - c_1 \sin x_3(t) \\ \frac{c_1}{x_2(t)}(u(t) - \cos x_3(t) \end{bmatrix} \tag{24}$$

and boundary conditions

$$\begin{array}{ll} x_1(0) = b_1, & x_1(t_f) = b_2 \\ x_2(0) = b_3, & x_2(t_f) = b_4 \\ x_3(0) = b_5, & x_3(t_f) = b_5. \end{array} \tag{25}$$

As with the previous example, we use the Hermite Simpson Separated collocation method and as such we will be using the same notation. For this example, though, we rewrite the equality conditions of 20 into the form:

$$\mathbf{C}(\mathbf{X},\mathbf{U},t_N) = \mathbf{A}\mathbf{X} + t_N\mathbf{B}\mathbf{F}(\mathbf{X},\mathbf{U}), \tag{26}$$

where $\mathbf{C}(\mathbf{X},\mathbf{U},t_N) \in \mathbb{R}^{(N-1)\times 3}$. We also use the unrolled form:

$$\mathbf{C}^\dagger(\mathbf{X}^\dagger,\mathbf{U},t_N) = \mathbb{A}\mathbf{X}^\dagger + t_N\mathbb{B}\mathbf{F}^\dagger(\mathbf{X}^\dagger,\mathbf{U}). \tag{27}$$

As in the previous example, we define our NLP decision vector as $\mathbf{z} = [\mathbf{X}^{\dagger T},\mathbf{U}^T,t_N]^T$. Our discretized problem is then

$$\min_{\mathbf{z}} t_N \tag{28}$$

subject to the equality constraints

$$\mathbf{C}^\dagger(\mathbf{z}) = \mathbf{0} \tag{29}$$

and the simple bounds

$$\mathbf{z}_{min} \le \mathbf{z} \le \mathbf{z}_{max}. \tag{30}$$

Now, in this example, we are concerned with building the Constraint Jacobian and the Lagrangian Hessian, which we will denote by $\bigtriangledown_{\mathbf{z}}\mathbf{C}^\dagger$ and $\bigtriangledown^2_{\mathbf{z}}L$, respectively. In building these, we take advantage of the fact that $\mathbf{F}^\dagger(\mathbf{X}^\dagger,\mathbf{U})$ is our only non-linear term, thus, *ADiGator* is only used to differentiate our dynamics file and we then build the Constraint Jacobian and Lagrangian Hessian using the result.

In order to build the Constraint Jacobian we first note that

$$\bigtriangledown_{\mathbf{z}}\,\mathbf{C}^\dagger = \begin{bmatrix} \bigtriangledown_{\mathbf{X}^\dagger}\mathbf{C}^\dagger & \bigtriangledown_{\mathbf{U}}\mathbf{C}^\dagger & \bigtriangledown_{t_N}\mathbf{C}^\dagger \end{bmatrix}, \tag{31}$$

where

$$\bigtriangledown_{\mathbf{X}^\dagger}\,\mathbf{C}^\dagger = \mathbb{A} + t_N\mathbb{B} \otimes \bigtriangledown_{\mathbf{X}^\dagger}\mathbf{F}^\dagger. \tag{32}$$

Here we note that we can take advantage of the way in which we write the unrolled Jacobian in order to perform the matrix multiplication $\mathbb{B} \otimes \bigtriangledown_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$. That is, $\mathbb{B} \in \mathbb{R}^{((N-1)*3)\times(N*3)}$, and $\bigtriangledown_{\mathbf{X}^\dagger}\mathbf{F}^\dagger \in \mathbb{R}^{(N*3)\times(N*3)}$, thus we can perform the matrix multiplication by reshaping $\bigtriangledown_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$ into a matrix of size $N \times (3*N*3)$, do the matrix multiplication, and then reshape the result into a matrix of size $((N-1)*3)\times(N*3)$. One should

note, though, that this only works if multiplying by a matrix on the left side. Similarly, we can compute $\bigtriangledown_{\mathbf{U}}\mathbf{C}^{\dagger}$ and $\bigtriangledown_{t_N}\mathbf{C}^{\dagger}$ as

$$\bigtriangledown_{\mathbf{U}}\ \mathbf{C}^{\dagger} = t_n\mathbb{B}\otimes\bigtriangledown_{\mathbf{U}}\mathbf{F}^{\dagger} \tag{33}$$

and

$$\bigtriangledown_{t_N}\ \mathbf{C}^{\dagger} = \mathbb{B}\mathbf{F}^{\dagger}. \tag{34}$$

We can do similar calculations in order to build the Lagrangian Hessian. First we start with the Lagrangian

$$L = t_N + \boldsymbol{\lambda}^T\left(\mathbb{A}\mathbf{X}^{\dagger} + t_N\mathbb{B}\mathbf{F}^{\dagger}\right). \tag{35}$$

For these calculations, we first define a new variable $\mathbf{Y} = [\mathbf{X}, \mathbf{U}]$, now, using some abusive notation, we may write our Lagrangian Gradient as

$$\bigtriangledown_{\mathbf{z}}\, L = \left[\ \bigtriangledown_{\mathbf{Y}^{\dagger}}L \quad \bigtriangledown_{t_N}L\ \right] \tag{36}$$

where

$$\bigtriangledown_{\mathbf{Y}^{\dagger}}\, L = \boldsymbol{\lambda}^T\left(\mathbb{A}\otimes[\mathbf{1},\mathbf{0}] + \mathbf{t_N}\mathbb{B}\otimes\bigtriangledown_{\mathbf{Y}^{\dagger}}\mathbf{F}^{\dagger}\right) \tag{37}$$

and

$$\bigtriangledown_{t_N}\, L = \boldsymbol{\lambda}^T\left([\mathbf{0},\mathbf{1}]^{\mathbf{T}} + \mathbb{B}\mathbf{F}^{\dagger}\right). \tag{38}$$

Now, we write our Lagrangian Hessian as

$$\bigtriangledown^2_{\mathbf{z}}\, L = \begin{bmatrix} \bigtriangledown^2_{\mathbf{Y}^{\dagger}}L & \bigtriangledown_{\mathbf{Y}^{\dagger}t_N}L \\ \bigtriangledown_{\mathbf{Y}^{\dagger}t_N}L & \bigtriangledown^2_{t_N}L \end{bmatrix}, \tag{39}$$

where,

$$\bigtriangledown^2_{\mathbf{Y}^{\dagger}}\, L = t_N\boldsymbol{\lambda}^T\mathbb{B}\otimes\bigtriangledown^2_{\mathbf{Y}^{\dagger}}\mathbf{F}^{\dagger}, \tag{40}$$

$$\bigtriangledown_{\mathbf{Y}^{\dagger}t_N}\, L = \boldsymbol{\lambda}^T\mathbb{B}\,\bigtriangledown_{\mathbf{Y}^{\dagger}}\,\mathbf{F}^{\dagger}, \tag{41}$$

and

$$\bigtriangledown^2_{t_N}\, L = 0. \tag{42}$$

The code for this example may be found in the directory `adigator/examples/optimization/minimumclimb`. In this directory there are four files which will solve the problem on three different meshes ($K = 10$, $K = 20$, and $K = 40$), where the initial guess for the second two iterations is calculated from the solution of the previous mesh. The files `main_1stderivs_nonvect` and `main_2ndderivs_nonvect` supply derivatives without using the vectorizated mode, while the files `main_1stderivs_vect` and `main_2ndderivs_vect` supply derivatives using the vectorizated mode. Furthermore, the function which computes the Constraint Jacobian (and calls the first dynamics derivative file) is given in `conswrap.m` and the function which computes the Lagrangian Hessian (and calls the second dynamics derivative file) is given in `laghesswrap.m`.

Here we see that, since we are only differentiating the dynamics file, that, when we use the vectorized mode, we must only create the derivative files a single time and they may be used to solve on as many different meshes as desired. If using the non-vectorized mode, however, the derivative files must be generated each time the value of K is changed. As far as performance goes, the user should note a slight increase in performance using the vectorized mode at the first derivative (the vectorized file should solve in 90-95% of the time it takes the non-vectorized). At the second derivative level, though, we see a major increase in performance as the vectorized file should solve in 5-10% of the time it takes the non-vectorized.

Appendix

In this user guide we employ the following notation. First, all scalars, vectors, and matrices are denoted by lower or upper case non-boldface (e.g., y or Y), lower case boldface (e.g., $\mathbf{y}$), and upper case boldface (e.g, $\mathbf{Y}$) symbols, respectively. Furthermore, if we are referring to a variable in MATLAB, we will generally use the typewriter text (e.g., `y, Y`). Thus, if $\mathbf{X}\in\mathbb{R}^{m\times n}$, then

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & \cdots & X_{1,n} \\ X_{2,1} & \cdots & X_{2,n} \\ \vdots & \ddots & \vdots \\ X_{m,1} & \cdots & X_{m,n} \end{bmatrix} \in \mathbb{R}^{m\times n}, \tag{43}$$

where $X_{i,j},\ (i=1,\ldots,m),\ (j=1,\ldots,n)$ are the *elements* of the $m \times n$ matrix $\mathbf{X}$. Similarly, the output of any matrix function of a matrix is denoted by a upper case bold letter. Consequently, if $\mathbf{F} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$ is a matrix function of the matrix variable $\mathbf{X}$, then $\mathbf{F}(\mathbf{X})$ has the form

$$\mathbf{F}(\mathbf{X}) = \begin{bmatrix} F_{1,1}(\mathbf{X}) & \cdots & F_{1,q}(\mathbf{X}) \\ F_{2,1}(\mathbf{X}) & \cdots & F_{2,q}(\mathbf{X}) \\ \vdots & \ddots & \vdots \\ F_{p,1}(\mathbf{X}) & \cdots & F_{p,q}(\mathbf{X}) \end{bmatrix} \in \mathbb{R}^{p\times q}, \tag{44}$$

where $F_{k,l}(\mathbf{X}),\ (k=1,\ldots,p),\ (l=1,\ldots,q)$ are the *elements* of the $p \times q$ matrix function $\mathbf{F}(\mathbf{X})$. The *Jacobian* of the matrix function $\mathbf{F}(\mathbf{X})$, denoted $\mathbf{JF}(\mathbf{X})$, is then a four-dimensional array of size $p \times q \times m \times n$ that consists of $pqmn$ elements. This multi-dimensional array will be referred to generically as the *rolled* representation of the derivative of $\mathbf{F}(\mathbf{X})$ with respect to $\mathbf{X}$ (where the term "rolled" is similar to the term "external" as used in [7]). In order to provide a more tractable form for the Jacobian of $\mathbf{F}(\mathbf{X})$, the matrix variable $\mathbf{X}$ and matrix function $\mathbf{F}(\mathbf{X})$ are transformed into the following so called *unrolled* form (where, again, the term "unrolled" is similar to the term "internal" [7]). First, $\mathbf{X} \in \mathbb{R}^{m\times n}$, is mapped isomorphically to a column vector $\mathbf{x}^\dagger \in \mathbb{R}^{mn}$, such that

$$x^\dagger_k = X_{i,j}, \quad \text{where} \quad k = i + m(j-1), \quad \forall \quad \begin{array}{rcl} i & = & 1,\ldots m \\ j & = & 1,\ldots n \\ k & = & 1,\ldots mn \end{array} . \tag{45}$$

Similarly, let $\mathbf{f}^\dagger(\mathbf{x}^\dagger) \in \mathbb{R}^{pq}$ be the one-dimensional transformation of the function $\mathbf{F}(\mathbf{X}) \in \mathbb{R}^{p\times q}$, such that

$$f^\dagger_k(\mathbf{x}^\dagger) = F_{i,j}(\mathbf{X}), \quad \text{where} \quad k = i + q(j-1), \quad \forall \quad \begin{array}{rcl} i & = & 1,\ldots q \\ j & = & 1,\ldots p \\ k & = & 1,\ldots qp \end{array} . \tag{46}$$

Here we note that the transformation from $\mathbf{X}$ to $\mathbf{x}^\dagger$ can be performed in MATLAB using the command `xdag = X(:)`, where `xdag` and `X` correspond to $\mathbf{x}^\dagger$ and $\mathbf{X}$, respectively. Furthermore, the MATLAB functions `sub2ind` and `ind2sub` are useful when switching between rolled and unrolled representations.

Using the one-dimensional representations $\mathbf{x}^\dagger$ and $\mathbf{f}^\dagger(\mathbf{x}^\dagger)$, the four-dimensional Jacobian, $\mathbf{JF}(\mathbf{X})$ can be represented in two-dimensional form as

$$\mathbf{Jf}^\dagger(\mathbf{x}^\dagger) = \begin{bmatrix} \frac{\partial f^\dagger_1}{\partial x^\dagger_1} & \cdots & \frac{\partial f^\dagger_1}{\partial x^\dagger_{mn}} \\ \frac{\partial f^\dagger_2}{\partial x^\dagger_1} & \cdots & \frac{\partial f^\dagger_2}{\partial x^\dagger_{mn}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f^\dagger_{pq}}{\partial x^\dagger_1} & \cdots & \frac{\partial f^\dagger_{pq}}{\partial x^\dagger_{mn}} \end{bmatrix} \in \mathbb{R}^{pq\times mn}. \tag{47}$$

Equation (47) provides what we refer to as the "unrolled Jacobian".

References

[1] M. J. Weinstein and A. V. Rao. A source transformation via operator overloading method for automatic differentiation in matlab. *ACM Transactions on Mathematical Software*, Submitted November 2013.

[2] M. A. Patterson, M. J. Weinstein, and A. V. Rao. An efficient overloaded method for computing derivatives of mathematical functions in matlab. *ACM Transactions on Mathematical Software,*, 39(3):17:1–17:36, July 2013.

[3] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Frontiers in Appl. Mathematics.* SIAM Press, Philadelphia, Pennsylvania, 2008.

[4] C. Bischof, B. Lang, and A. Vehreschild. Automatic differentiation for MATLAB programs. *Proceedings in Applied Mathematics and Mechanics*, 2(1 Joh Wiley):50–53, 2003.

[5] G Wanner and E Hairer. *Solving Ordinary Differential Equations II*, volume 1. Springer-Verlag, Berlin, 1991.

[6] T. Burg, D. Dawson, and P. Vedagarbha. A redesigned dcal controller without velocity measurements: theory and demonstration. *Robotica*, 15:337–346, 5 1997.

[7] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222, April–June 2006.