AIAA/AAS Astrodynamics Specialist Conference
13 - 16 August 2012, Minneapolis, Minnesota

AIAA 2012-4521

A Method For Computing Derivatives in MATLAB

Michael. A. Patterson[*]
Matthew Weinstein[†]
Anil V. Rao[‡]

University of Florida
Gainesville, FL 32611

ABSTRACT

An object-oriented method is presented that generates derivatives of functions defined by MATLAB computer codes. The method uses operator overloading together with forward mode automatic differentiation and produces a new MATLAB code that computes the derivatives of the outputs of the original function with respect to the variables of differentiation. The method can be used recursively to generate derivatives of any order that are desired and has the feature that the derivatives of a function are generated simply by evaluating the function on an instance of the class. The method is described briefly and is demonstrated on an example.

I. Introduction

In the past few decades, a great deal of research has focused on *automatic differentiation* (AD). AD is the process of determining accurate derivatives of a function defined by computer programs[1] using the rules of differential calculus. In other words, the goal of AD is to employ ordinary calculus algorithmically with the goal of obtaining machine precision accuracy derivatives in a computationally efficient manner. AD exploits the fact that a computer code that implements a general function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ can be decomposed into a sequence of elementary function operations. The derivative is then obtained by applying the standard differentiation rules (e.g., product, quotient, and chain rules). The most well known methods for automatic differentiation are *forward and reverse mode*. In either forward or reverse mode, each link in the calculus chain rule is implemented until the derivative of the input with respect to itself is encountered. The fundamental difference between forward and reverse modes is the direction in which the operations are performed. In forward mode, the operations are performed from the variable of differentiation to the final derivative of the function, while in reverse mode the operations are performed from the function back to the variable of differentiation.

In this paper we present a new approach for automatic differentiation of MATLAB code. The method of this paper combines features of operator-overloading and source transformation together

[*]Ph.D. Candidate, Department of Mechanical and Aerospace Engineering. E-mail: mpatterson@ufl.edu.
[†]Ph.D. Candidate, Department of Mechanical and Aerospace Engineering, E-mail: weinstein87@gmail.com.
[‡]Associate Professor, Department of Mechanical and Aerospace Engineering. E-mail: anilvrao@ufl.edu. Associate Fellow, AIAA. Corresponding Author.

with forward mode automatic differentiation in a manner that produces a new MATLAB code that contains statements to compute the derivative of the original MATLAB function in terms of the native MATLAB library. In order to realize our approach, a new MATLAB object class, called *CADA* (where *CADA* stands for "Computation of Analytic Derivatives Automatically"), is developed. Using instances of class *CADA*, it is possible to obtain the derivatives by *evaluating* the function on the input *CADA* object. *CADA* then produces a file that contains the statements that compute the derivative of the original function with respect to the input variable. The sequence of mathematical operations in the file that computes the derivative file is identical to those obtained by applying the forward mode on a numeric value of the variable of differentiation. Finally, the method is applied to an example to demonstrate its utility.

II. Forward Mode Automatic Differentiation

Consider a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$. Assume that $\mathbf{f}(\mathbf{x})$ can be reduced to a sequence of elementary function operations (e.g., polynomials, exponential functions, trigonometric functions). Furthermore, suppose that a computer implementation of the $\mathbf{f}(\mathbf{x})$ has been written in the MATLAB programming language. The goal is to automatically determine $\mathbf{Jf}(\mathbf{x})$, that is, the goal is to determine the derivative of each output function f_{ij}, $(i = 1, \ldots, p)$, $(j = 1 \ldots, q)$ with respect to each input element x_{kl}, $(k = 1, \ldots, m)$, $(l = 1, \ldots, n)$. Moreover, it is desired to obtain a new MATLAB code that implements these derivatives for future use. The method of this paper uses an object-oriented approach that implements forward mode automatic differentiation in such a way that each operation in a function is differentiated in terms of the results of previous operations, and the result of each operation is printed to a file. In forward mode differentiation, the chain rule is used repeatedly until the step in the chain is encountered where the derivative of the outermost function with respect to the input is obtained. In the case of a scalar function $y = f(g(x))$, the derivative $\partial f/\partial x$ is obtained via the chain rule as $\partial f/\partial x = \partial f/\partial g \cdot \partial g/\partial x$. Finally, as part of the method developed in this paper, only the nonzero elements of the Jacobian are stored, reducing memory requirements and taking advantage of Jacobian sparsity.

A. *CADA* Differentiation Method

The method used in *CADA* to differentiate a function by evaluating the function on overloaded *CADA* objects relies upon the interaction between the different *CADA* object properties and a global *CADA* environment, GLOBALCADA. Each *CADA* object contains a **function** and **derivative** property, where the **function** property contains sizing and naming information, and the **derivative** property contains *nonzero* derivative locations. When any mathematical operation is evaluated on a *CADA* object, the lines of code that represent the mathematical operation and the derivative of the mathematical operation are printed to a temporary file. As an example of how an overloaded operation is written to the temporary file, consider a mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and assume that this mathematical operation is applied to a *CADA* object $\mathcal{X}$, resulting in an output *CADA* object $\mathcal{Z}$ (that is, $\mathcal{Z} = \mathbf{f}(\mathcal{X})$). While the object properties of $\mathcal{X}$ contain information from previous overloaded evaluations up to that point in the overall function, the GLOBALCADA information is needed to define the function and derivative handles of the output $\mathcal{Z}$. First, a check is performed to determine if the input $\mathcal{X}$ is a *CADA* numeric object. In this case the **function** property of $\mathcal{Z}$ is calculated directly and no code is written to the temporary file. If $\mathcal{X}$ is not a *CADA* numeric object, new function and derivative handles are defined for $\mathcal{Z}$ and the code that represents $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and the derivative of $\mathbf{z} = \mathbf{f}(\mathbf{x})$ are printed to the temporary file. In this second case, only the code that corresponds to the *nonzero* derivatives of the operation with respect to each *CADA* instance is written to the temporary file. Because the derivatives are computed sparsely, it is necessary that the appropriate elements of the **function** property of $\mathcal{X}$ are properly referenced such that the code resulting from the application of the calculus chain rule is applied to only the elements in the array whose derivatives are not zero. For every overloaded operation he GLOBALCADA properties are also updated to track the occurrence of variables in the resulting derivative code (see Ref. 2 for details).

B. *CADA* Setup and Environment Instantiation

The *CADA* global environment and all *CADA* instances that are used in generating derivative code are instantiated using the function **cadasetup**. The **cadasetup** function performs the following operations: (1) instantiates the global structure GLOBALCADA; (2) defines the initial property values for GLOBALCADA; (3) creates and opens a temporary file to which the derivative code will be written; (4) and creates all instances that will be used during the *CADA* session. For each *CADA* instance that is desired, **cadasetup** requires the following four inputs: a string containing the name of the *CADA* instance (where each instance must have a unique name and the name cannot be a MATLAB reserved word); two integers that, respectively, define the row and column size of the *CADA* instance; and an integer flag that indicates whether or not *CADA* should compute derivatives with respect to the particular *CADA* instance. If the derivative option flag corresponding to a *CADA* instance is unity, then derivatives with respect to the *CADA* instance are calculated. On the other hand, if the derivative option flag corresponding to a *CADA* instance is zero, derivatives with respect to the *CADA* instance are not calculated.

C. Differentiation of Unary Mathematical Functions

The overloaded versions of all standard unary mathematical functions (e.g., polynomial, trigonometric, exponential) operate on only a single *CADA* input object, resulting in an output that is a new *CADA* object. Consider an arbitrary unary mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and assume that this mathematical operation is applied to a *CADA* object $\mathcal{X}$, resulting in an output *CADA* object $\mathcal{Z}$ (that is, $\mathcal{Z} = \mathbf{f}(\mathcal{X})$). If the input *is not* a *CADA* numeric object, new code that represents the function and its derivative with respect to each *CADA* instance is written to the temporary file. The nonzero derivative indices of the output will be the same as the nonzero derivative indices of the input. If the input to the overloaded unary operation is a *CADA* numeric object, the output of the overloaded unary operation is also a *CADA* numeric object. In this case the numeric array in the *CADA* numeric object is evaluated and neither are derivatives calculated nor is code written to the temporary file.

D. Differentiation of Binary Mathematical Functions

The overloaded binary mathematical functions (e.g., plus, minus, times, array multiplication) have two inputs, where at least one of the inputs must be a *CADA* object and the output is a *CADA* object. Consider a binary mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x}, \mathbf{y})$. If $\mathbf{x}$ is a *CADA* object but $\mathbf{y}$ is not a *CADA* object, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathcal{X}, \mathbf{y})$. Similarly, if $\mathbf{y}$ is a *CADA* object but $\mathbf{x}$ is not a *CADA* object, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathbf{x}, \mathcal{Y})$. Finally, if both $\mathbf{x}$ and $\mathbf{y}$ are *CADA* objects, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathcal{X}, \mathcal{Y})$. First, it is necessary to determine the class of each input. If only one of the inputs is a *CADA* object, then the operation follows in a manner similar to that of a unary function. In the case where both inputs are *CADA* objects, it is necessary to determine if any or both inputs are *CADA* numeric objects. If both inputs are *CADA* numeric objects, the function is evaluated on the numeric values of $\mathcal{X}$ and $\mathcal{Y}$, resulting in the *CADA* numeric object $\mathcal{Z}$. Furthermore, as is the case with all *CADA* numeric objects, no derivatives are calculated and no code is written to the temporary file. If exactly one of the two inputs is a *CADA* numeric object, the numeric value of the *CADA* numeric object is found and the overloaded operation is performed as if only one of the inputs is a *CADA* object. Lastly, if both *CADA* objects contain derivatives with respect to the same *CADA* instance, the binary chain rule (e.g., product rule, quotient rule, etc.) is applied to calculate the nonzero derivatives. The nonzero derivative indices of the output are determined by taking the union of the nonzero derivative indices of $\mathcal{X}$ and $\mathcal{Y}$.

III. Example

We now apply *CADA* to an example. The example is a large sparse nonlinear programming problem (NLP). The objective of the example is to demonstrate how *CADA* generates the constraint Jacobian and Lagrangian Hessian of a complex NLP. In the example the efficiency *CADA* is compared against *INTLAB*, *ADMAT*, and *MAD*. Finally, it is noted that all computations shown in this section were performed using an Apple MacPro 2.26 GHz Quad-Core Intel Xeon computer with 16 GB of RAM running Mac OS-X Snow Leopard version 10.6.8 and MATLAB R2010b.

Consider the following sparse nonlinear programming problem (NLP) that arises from the discretization of the optimal control problem from Ref. 3 using the Radau pseudospectral method described in Refs. 4–6. The NLP decision vector $\mathbf{z} \in \mathbb{R}^{6N+4}$ is given as

$$\mathbf{z} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{w}_1, \mathbf{w}_2), \tag{1}$$

where N is a parameter that defines the total number of *Legendre-Gauss-Radau* (LGR) points,[7] $\mathbf{x}_i = (x_{1,i}, \ldots, x_{N+1,i})$, $(i = 1, \ldots, 4)$, and $\mathbf{w}_i = (w_{1,i}, \ldots, w_{N,i})$, $(i = 1, 2)$. The objective of the NLP is to minimize the cost function

$$J = -x_{N+1,1} \tag{2}$$

subject to the nonlinear algebraic constraints

$$\begin{array}{ll}
\displaystyle\sum_{k=1}^{N+1} D_{i,k} x_{k,1} - \frac{\kappa}{2} x_{i,3} = 0, & \displaystyle\sum_{k=1}^{N+1} D_{i,k} x_{k,2} - \frac{\kappa}{2} \frac{x_{i,4}}{x_{i,1}} = 0, \\
\displaystyle\sum_{k=1}^{N+1} D_{i,k} x_{k,3} - \frac{\kappa}{2} \left[\frac{x_{4,i}^2}{x_{i,1}} - \frac{1}{x_{i,1}^2} + K_i w_{i,1} \right] = 0, & \displaystyle\sum_{k=1}^{N+1} D_{i,k} x_{k,4} - \frac{\kappa}{2} \left[-\frac{x_{3,i} x_{4,i}}{x_{i,1}} + K_i w_{i,2} \right] = 0, \\
w_{i,1}^2 + w_{i,2}^2 - 1 = 0, &
\end{array} \tag{3}$$

and the equality constraints

$$x_{1,1} - u_1 = 0\,,\ x_{1,2} = 0\,,\ x_{1,3} = 0\,,\ x_{1,4} - u_2 = 0\,,\ x_{N+1,2} = 0\,,\ \sqrt{u_3 / x_{N+1,1}} - x_{N+1,4} = 0, \tag{4}$$

where $i = 1, \ldots, K$ in Eq. (3) and $D_{i,k}$, $(i = 1, \ldots, N,\ k = 1, \ldots, N+1)$ is the Radau pseudospectral differentiation matrix,

$$t_i \quad = \quad \frac{\kappa}{2} s_i + \frac{\kappa}{2} \quad , \quad K_i \quad = \quad \frac{u_4}{u_5 t_i + u_6}. \tag{5}$$

$(s_1, \ldots, s_N)$ are the N LGR points on the interval $[-1, +1)$ and $s_{N+1} = +1$ and $(i = 1, \ldots, N)$ in Eqs. (3) and (5). The total number of LGR points, $N = N_k K$, is obtained by dividing the problem into K mesh intervals using N_k LGR points in each mesh interval (for details see Ref. 6). The numerical values of the parameters $u_1, \ldots, u_6$, and κ used in this example are given in Table 1.

The NLP defined in Eqs. (2) and (3) can be written in the more generic mathematical form

$$\text{minimize } f(\mathbf{z}) \text{ subject to } \mathbf{g}(\mathbf{z}) = \mathbf{0} \text{ and } \mathbf{z}_{\min} \leq \mathbf{z} \leq \mathbf{z}_{\max}. \tag{6}$$

Table 1: Numerical Values of Parameters for Example 2.

Parameter	u_1	u_2	u_3	u_4	u_5	u_6	κ
Value	1	1	1	0.1405	-0.0749	1	3.32

We divide the analysis in this example into two parts. First, we study the efficiency with which *CADA* generates and computes the constraint Jacobian, $\partial \mathbf{g}/\partial \mathbf{z}$, and the Lagrangian Hessian, $\partial^2 L/\partial \mathbf{z}^2$, where $L = \sigma f(\mathbf{z}) + \boldsymbol{\lambda}^{\mathsf{T}}\mathbf{g}(\mathbf{z})$ and the variables $\sigma \in \mathbb{R}$ and $\boldsymbol{\lambda} \in \mathbb{R}^{6N}$ are the Lagrange multipliers for the cost and the constraints, respectively [where we note that $\boldsymbol{\lambda} \in \mathbb{R}^{5N}$ because the NLP contains $5N$ nonlinear constraints as shown in Eq. (3)]. Second, we demonstrate the efficiency of *CADA* in solving the NLP for various values of N. In order to incorporate both first and second derivatives, in this research the NLP was solved using the second-derivative open-source NLP solver *IPOPT*[8] with the indefinite sparse symmetric linear solver MA57.[9] As a second derivative NLP solver, *IPOPT* requires the user to supply the objective function gradient, $\partial f/\partial \mathbf{z}$, the constraint Jacobian, $\partial \mathbf{g}/\partial \mathbf{z}$, the Lagrangian Hessian, $\partial^2 \mathcal{L}/\partial \mathbf{z}^2$, the sparsity pattern of the constraint Jacobian, and the sparsity pattern of the lower-triangular portion of the Lagrangian Hessian.

In order to analyze the derivative efficiency as the NLP grows in size, we fix the number of LGR points in each mesh interval to four (that is, $N_k = 4$) and we vary the number of mesh intervals K. Thus, in this example the total number of LGR points is $N = 4K$. Table 2 shows the Jacobian-to-function ratio, CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$), for the NLP constraint function of Eq. (6 as a function of N. It is seen that the ratio CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) using *CADA* is smaller than any of the other methods for all values of N except $N = 256$. In the case $N = 256$, *INTLAB* is more efficient than *CADA* in generating the constraint Jacobian, but *CADA* is more efficient than either *ADMAT* or *MAD*. Table 3 shows the Hessian-to-function ratio, CPU($\mathbf{H}L$)/CPU(L), for the Lagrangian Hessian $L = \sigma f(\mathbf{z}) + \boldsymbol{\lambda}^{\mathsf{T}}\mathbf{g}(\mathbf{z})$. In the case of the Hessian Lagrangian it is seen that *CADA* is more efficient than *INTLAB* and is *significantly* more efficient than either *ADMAT* or *MAD* for all values of N.

Table 2: *CADA* Derivative Generation Time and Jacobian-to-Constraint Computation Time Ratio, CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$), for Example Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. Derivative Generation Times Using *CADA* Were Averaged Over 100 File Generations, While CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) Was Obtained by Averaging the Values Obtained Over 1000 Jacobian and Function Evaluations.

K	$N = 4K$	Derivative Code Generation Time Using *CADA* (s)	CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) Ratio Using *CADA*	CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) Ratio Using *INTLAB*	CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) Ratio Using *ADMAT*	CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) Ratio Using *MAD*
8	32	0.279	5.26	31.3	37.4	48.2
16	64	0.468	6.83	32.4	37.6	47.8
32	128	0.854	9.24	33.1	37.8	47.5
64	256	1.62	15.1	34.1	46.9	48.2
128	512	3.18	25.6	38.8	47.1	50.4
256	1024	6.28	44.5	40.1	48.1	51.6

We now turn our attention to the solution of the aforementioned NLP using *CADA*, *INTLAB*, *ADMAT*, and *MAD* with the NLP solver *IPOPT*. Next, because *IPOPT* requires that the user provide the constraint Jacobian and Lagrangian Hessian sparsity patterns, it was necessary to construct these patterns prior to solving the NLP. While it is possible to determine the Jacobian and Lagrangian Hessian sparsity patterns with *ADMAT*, we found the process for determining the Hessian sparsity pattern to be unacceptably slow. In addition, *INTLAB* does not have a built-in capability of determining either Jacobian or Hessian sparsity patterns. As a result, the approach used in this research was to use the constraint Jacobian and Lagrangian Hessian sparsity pat-

Table 3: *CADA* Derivative Generation Time and Lagrangian Hessian to Lagrangian Computation Time Ratio, CPU($\mathbf{H}L$)/CPU(L) for Example Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. Derivative Generation Times Using *CADA* Were Averaged Over 100 File Generations, While CPU($\mathbf{H}L$)/CPU(L) Was Obtained by Averaging the Values Obtained Over 1000 Lagrangian Hessian and Lagrangian Function Evaluations.

K	$N = 4K$	Hessian Code Generation Time Using *CADA* (s)	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *CADA*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *INTLAB*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *ADMAT*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *MAD*
8	32	0.606	17.52	119.4	771.3	588.4
16	64	0.886	28.93	156.4	1072	906.7
32	128	1.46	65.78	411.4	1831	2334
64	256	2.67	242.4	1365	3476	7456
128	512	5.03	835.2	4058	9592	23157
256	1024	10.27	2806	15553	33623	89330

terns that *CADA* generates simultaneous with its generation of the Jacobian and Hessian derivative functions. These sparsity patterns were then supplied to *IPOPT* for use with all three automatic differentiation programs. Finally, it is noted that the constraint Jacobian was determined by differentiating Eq. (3). On the other hand, it is known that the terms Eq. (3) involving the coefficients $D_{i,k}$, $(i = 1, \ldots, N,\ j = 1, \ldots, N+1)$ are *linear* in the decision vector. Consequently, the second derivatives of these terms are zero and, thus, will not appear in the Lagrangian Hessian. Thus, the terms involving the coefficients $D_{i,k}$, $(i = 1, \ldots, N,\ j = 1, \ldots, N+1)$ are not included in the computation of the NLP Lagrangian.

The NLP given in Eqs. (2)–(4) was solved using *IPOPT* for $K = (8, 16, 32, 64, 128, 256)$ with $N_k = 4$, $(k = 1, \ldots, K)$ (that is, the same number of Legendre-Gauss-Radau points[7] was used in every mesh interval) with the following initial guess: $\{x_{1,i}, \cdots, x_{N+1,i}\} = 1,\ i = 1, 4$, $\{x_{1,i}, \cdots, x_{N+1,i}\}$ $= 0,\ i = 2, 3$, and $\{w_{1,i}, \ldots, x_{N,i}\} = 0,\ i = 1, 2$. The results obtained for this example are shown in Tables 4 and 5. As may be expected, the derivative code generation time using *CADA* increases with K. In addition, it is seen that the time required to solve the NLP (excluding the time required to create the derivative files) using *CADA* is smaller than the corresponding computation time required by either *INTLAB* or *ADMAT*. Next, it is noted that the total computation time (that is, the derivative code generation time plus the time to solve the NLP) using derivatives obtained by *CADA* is less than the solution time required by either of the other automatic differentiators, and for the larger values of K the total computation time using *CADA* is significantly *less* than either *INTLAB*, *ADMAT*, or *MAD*. Thus, *CADA* becomes more computationally attractive as the size of the NLP increases.

Table 4: *IPOPT* Solution Times for Example Using *CADA*, *INTLAB*, *ADMAT*, and *MAD*.

K	$N = 4K$	Derivative Code Generation Time Using *CADA* (s)	IPOPT Run Time Using *CADA* (s)	IPOPT Run Time Using *INTLAB* (s)	IPOPT Run Time Using *ADMAT* (s)	IPOPT Run Time Using *MAD* (s)
8	32	0.893	0.491	1.68	6.37	5.31
16	64	1.36	0.834	2.28	10.39	8.17
32	128	2.33	1.95	5.46	18.51	28.62
64	256	4.30	5.06	18.77	45.67	101.1
128	512	8.22	19.24	67.64	225.3	368.1
256	1024	16.56	117.2	423.1	1386	2298

Table 5: Problem Size and Densities of Constraint Jacobian and Lagrangian Hessian for Example.

K	$N = 4K$	NLP Variables	NLP Constraints	Jacobian Non-Zeros	Jacobian Density (%)	Hessian Non-Zeros	Hessian Density (%)
8	32	196	161	994	3.15	321	0.835
16	64	388	321	1986	1.59	641	0.425
32	128	772	641	3970	0.802	1281	0.215
64	256	1540	1281	7938	0.402	2561	0.108
128	512	3076	2561	15874	0.201	5121	0.054
256	1024	6148	5121	31746	0.101	10241	0.027

IV. Conclusions

An object-oriented method has been described for computing derivatives of MATLAB functions. A new class called *CADA* has been developed. The derivatives of a general function can be obtained by evaluating a function of interest on a *CADA* object and writing the derivatives to a file that itself can be executed. The derivative file contains the same sequence of operations as the forward mode of automatic differentiation. Moreover, because the derivative function has the same input as the original function computer code, it is possible to apply the method recursively to generate higher-order derivatives. As a result, the method provides the ease of use associated with forward mode automatic differentiation while simultaneously providing the same functionality as would be obtained had the analytic derivatives been coded by hand. The key components of the method have been described in detail and the usability of the method was demonstrated on an example. The method presented in this paper provides an efficient and reliable approach for computing analytic derivatives of general MATLAB functions.

Acknowledgments

The authors gratefully acknowledge support for this research from the U.S. Office of Naval Research (ONR) under Grant N00014-11-1-0068 and from the U.S. Defense Advanced Research Projects Agency (DARPA) Under Contract HR0011-12-0011.

Disclaimer

The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

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

[2]Patterson, M. A., Weinstein, M., and Rao, A. V., "An Efficient Overloaded Method for Computing Derivatives of Mathematical Functions in MATLAB," *ACM Transactions on Mathematical Software,* In Revision, April 2012. Current Manuscript Available at http://vdol.mae.ufl.edu/SubmittedJournalPublications/cada-TOMS-2011-0055-Revision-April-2012.pdf.

[3]Bryson, A. E. and Ho, Y.-C., *Applied Optimal Control*, Hemisphere Publishing, New York, 1975.

[4]Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T., "A Unified Framework for the Numerical Solution of Optimal Control Problems Using Pseudospectral Methods," *Automatica*, Vol. 46, No. 11, November 2010, pp. 1843–1851.

[5]Garg, D., Patterson, M. A., Darby, C. L., Francolin, C., Huntington, G. T., Hager, W. W., and Rao, A. V., "Direct Trajectory Optimization and Costate Estimation of Finite-Horizon and Infinite-Horizon Optimal Control Problems via a Radau Pseudospectral Method," *Computational Optimization and Applications*, Vol. 49, No. 2, June 2011, pp. 335–358.

[6]Patterson, M. A. and Rao, A. V., "Exploiting Sparsity in Direct Collocation Pseudospectral Methods for Solving Continuous-Time Optimal Control Problems," *Journal of Spacecraft and Rockets,*, Vol. 49, No. 2, March–April 2012, pp. 364–377.

[7]Abramowitz, M. and Stegun, I., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Dover Publications, New York, 1965.

[8]Biegler, L. T. and Zavala, V. M., "Large-Scale Nonlinear Programming Using IPOPT: An Integrating Framework for Enterprise-Wide Optimization," *Computers and Chemical Engineering*, Vol. 33, No. 3, March 2008, pp. 575–582.

[9]Duff, I. S., "MA57—a Code for the Solution of Sparse Symmetric Definite and Indefinite Systems," *ACM Transactions on Mathematical Software*, Vol. 30, No. 2, 2004, pp. 118–144.