NASA/TM–2014-218536

A Walsh Function Module Users' Manual

Peter A. Gnoffo
Langley Research Center, Hampton, Virginia

October 2014

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2014-218536

A Walsh Function Module Users' Manual

Peter A. Gnoffo
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2014

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

The solution of partial differential equations (PDEs) with Walsh functions offers new opportunities to simulate many challenging problems in mathematical physics. The approach was developed to better simulate hypersonic flows with shocks on unstructured grids. It is unique in that integrals and derivatives are computed using simple matrix multiplication of series representations of functions without the need for divided differences. The product of any two Walsh functions is another Walsh function - a feature that radically changes an algorithm for solving PDEs. A FORTRAN module for supporting Walsh function simulations is documented. A FORTRAN code is also documented with options for solving time-dependent problems: an advection equation, a Burgers equation, and a Riemann problem. The sample problems demonstrate the usage of the Walsh function module including such features as operator overloading, Fast Walsh Transforms in multi-dimensions, and a Fast Walsh reciprocal.

Contents

1 Introduction **4**

2 Walsh Function Module: WALSH_TOOLS **6**

2.1 Parameters and Arrays . . . 6
2.1.1 dp . . . 6
2.1.2 gns . . . 6
2.2 Derived Types . . . 6
2.2.1 type(g) . . . 6
2.2.2 type(walsh) . . . 7
2.3 Operator Overloading . . . 9
2.3.1 assignment(=) . . . 9
2.3.2 operator(+) . . . 9
2.3.3 operator(-) . . . 10
2.3.4 operator(*) . . . 10
2.3.5 operator(**) . . . 10
2.3.6 operator(/) . . . 10
2.4 Functions . . . 11
2.4.1 f_from_comp . . . 11
2.4.2 absw . . . 11
2.4.3 sqrtw . . . 11
2.4.4 gn . . . 11
2.4.5 pmap . . . 12
2.4.6 intx . . . 12
2.4.7 inty . . . 12
2.4.8 intz . . . 12
2.4.9 intt . . . 13
2.5 Subroutines . . . 13
2.5.1 allocate_walsh . . . 13
2.5.2 deallocate_walsh . . . 13
2.5.3 initialize_walsh . . . 13
2.5.4 fast_walsh_gather_4d . . . 14
2.5.5 g_hat_bisection . . . 15
2.5.6 g_recursion . . . 15
2.5.7 check_walsh . . . 15
2.5.8 setup_domain . . . 16

3 Boundary Module: BOUNDARY **17**

3.1 **overlap** . . . 17
3.2 profile . . . 19
3.3 profile_burgers . . . 20
3.4 profile_sod . . . 20
3.5 boundary_from_interior . . . 20

4 LAPACK Module: LAPACK **23**

5 Demonstration Program: DEMO **24**
5.1 Test Cases . . . 24
5.1.1 Advection . . . 24
5.1.2 Burgers Equation . . . 25
5.1.3 Riemann Problem . . . 26
5.2 Compilation . . . 28
5.2.1 Makefile.env . . . 28
5.2.2 make.dependencies . . . 28
5.2.3 Makefile.rules . . . 29
5.2.4 Makefile . . . 29
5.3 Inputs . . . 29
5.3.1 **demo_code** . . . 29
5.3.2 **nu** . . . 29
5.3.3 **p_alpha** . . . 30
5.3.4 **p_tau** . . . 30
5.3.5 **p_domain** . . . 30
5.3.6 **overlap_x** . . . 30
5.3.7 **overlap_t** . . . 30
5.3.8 **dt** . . . 30
5.3.9 **t_max** . . . 31
5.3.10 **truncate** . . . 31
5.4 OutputFiles . . . 31
5.4.1 Screen Output . . . 31
5.4.2 Plot Files . . . 31
5.5 Flowchart . . . 32

6 Sample Solutions **34**
6.1 Advection . . . 34
6.2 Burgers Equation . . . 36
6.3 Riemann Problem . . . 39

7 Summary **41**

1 Introduction

The solution of partial differential equations (PDEs) with Walsh functions offers new opportunities to simulate many challenging problems in mathematical physics. The approach was developed to better simulate hypersonic flows with shocks on unstructured grids. [1]

The Walsh function approach is unique in that integrals and derivatives are computed using simple matrix multiplication of series representations of functions without the need for divided differences. The product of any two Walsh functions is another Walsh function - a feature that radically changes an algorithm for solving PDEs.

While this algorithm holds great promise, there are coding infrastructure challenges that make it difficult to further explore and develop numerical simulation tools using Walsh functions. Some unique strengths of Walsh functions are the capabilities to systematically account for non-linear interactions of component waves and to explicitly propagate boundary conditions across the solution domain. However, the implementation of these capabilities presents tedious programming challenges. Multiplication and division of two Walsh functions are fundamentally different from any currently supported operations in any programming language. Keeping track of Jacobians required for the solution of systems of PDEs can become especially tedious in this environment. Fortunately, all of this tedious detail can be supported in the background through use of a module that supports operator overloading and other commonly used functions and routines in the solution of partial differential equations. This support greatly simplifies the learning curve for new users wanting to explore the use of Walsh functions in solving PDEs.

The purpose of this manual is to document the Walsh function support infrastructure in FORTRAN module WALSH_TOOLS. It also provides three demonstration problems providing examples of how the WALSH_TOOLS infrastructure is used to solve PDEs. It is assumed the reader/user has some familiarity with FORTRAN 2003 in general and the use of derived types in particular. This manual is intended to be used in association with the original documentation [1] of the algorithm. Only the interface to Walsh function operations and algorithms is described; the reader must refer to the original documentation as well as the FORTRAN program files described herein for details of implementation.

Two algorithms not previously documented are supported within the module WALSH_TOOLS.

- The Fast Walsh Transform (FWT) in multi-dimensions is key to efficient solution of problems using a large number (N) of terms in the series. An element of any solution algorithm requires the transformation from wave number representation to discrete value representation and back. The FWT implements this transformation in order $N\log(N)$ operations (if $N = 2^p$) rather than order N^2 operations as documented previously. [1]

- The Fast Walsh Reciprocal (FWR) provides the inverse of a Walsh symmetric matrix in order $N\log(N)$ operations. Any problems involving the evaluation of the reciprocal of a function requires evaluation of a Walsh symmetric matrix inverse.

Documentation and derivation of the FWT and FWR algorithms are planned for an upcoming paper. For now, the implementation of these algorithms is documented in the accompanying FORTRAN code.

The remainder of this manual is organized as follows. Section 2 documents the Walsh function specific oerations, functions, and subroutines that are intended to provided infrastructure support for new research involving Walsh functions in the solution of PDEs. Section 3 documents the

routines used to define boundary conditions and exact solutions to the demonstration problems. Section 4 briefly describes the linear solver software routines [2] used to solve the PDEs with Newton relaxation. Section 5 defines the three demonstration test problems and documents how module WALSH_TOOLS is used obtain their solution. Finally, Section 6 documents results of the problems defined in Sec. 5. The ultimate goal of this effort is to encourage new users to borrow and improve upon the algorithms documented here to further advance the use of Walsh functions in the solution of non-linear, partial differential equations.

2 Walsh Function Module: WALSH_TOOLS

The Walsh function module WALSH_TOOLS includes all derived types, procedures, functions, and subroutines required to define and use Walsh functions in the solution of partial differential equations. It is assumed that users will want to start with the current code structure and then make changes associated with their own research and application needs. Only variables, functions, and routines declared public - intended for general use - are documented here. Note that the Jacobians of every variable of type **walsh** are automatically computed.

2.1 Parameters and Arrays

2.1.1 dp

```
integer,  parameter :: dp  = selected_real_kind(15, 307)
```

All real values in all modules are set to type **real(dp)**. The parameter **dp** allows the programmer to modify the real precision throughout. Double precision is the default.

2.1.2 gns

```
type(g), dimension(:), allocatable :: gns
```

The array of orthonormal Walsh functions created in subroutine g_hat_bisection or subroutine gn_recursion.

2.2 Derived Types

2.2.1 type(g)

```
type g
  integer,  dimension(:), pointer :: segment
  integer,  dimension(:), pointer :: factor
  integer                         :: level
end type
```

Each orthonormal Walsh function g_n is of **type(g)**. This derived type includes three essential elements for describing the nth Walsh function.

- The one-dimensional integer array **segment** is allocated n and includes the dimensionless segment lengths (1 or 2) corresponding to $\hat{g}_n$ of Eq. 6 of Reference[1].
- The one-dimensional integer array **factor** is allocated n. The mth element of **factor** defines the mapped location k of the product $g_n g_m = g_k$. This array corresponds to the nth column of the multiplication table $\mathcal{P}(n, m)$ of Table 2 in Reference [1].
- The integer **level** corresponds to the value of p in Eq. (3) of Reference[1]. Segment size for g_n is determined by p in Eq. (2) of Reference[1].

2.2.2 type(walsh)

```
type walsh
  integer                                  :: len
  integer                                  :: ndep
  integer                                  :: nbcx
  integer                                  :: nbcy
  integer                                  :: nbcz
  integer                                  :: nbct
  logical                                  :: jacb
  integer,  dimension(4)                   :: nseg
  real(dp), dimension(4,2)                 :: domain
  logical,  dimension(:    ), allocatable :: bdep
  logical,  dimension(:    ), allocatable :: bcx
  logical,  dimension(:    ), allocatable :: bcy
  logical,  dimension(:    ), allocatable :: bcz
  logical,  dimension(:    ), allocatable :: bct
  real(dp), dimension(:    ), allocatable :: comp
  real(dp), dimension(:,:,:), allocatable :: jac
  real(dp), dimension(:,:,:), allocatable :: jacbcx
  real(dp), dimension(:,:,:), allocatable :: jacbcy
  real(dp), dimension(:,:,:), allocatable :: jacbcz
  real(dp), dimension(:,:,:), allocatable :: jacbct
end type
```

Every dependent variable in the solution of a system of PDEs is of type **walsh**. This derived type includes all of the information required to evaluate all wave components and Jacobians across a domain for most unary and binary operations (+,-,*,/,**, $\int$,and $\frac{\partial()}{\partial(x,y,z,t)}$) involving operands of type **walsh**.

- The integer **len** equals the maximum number of segments in the domain. This value is inherited by any derived function of type **walsh**.
- The integer **ndep** is the total number of primary, dependent variables in the domain. This value is inherited by any derived function of type **walsh**. If the domain is global then **ndep** is usually equal to the number of PDEs to be solved. If the domain is a boundary then **ndep** equals the total number of boundary conditions in the respective coordinate direction.
- The integer **nbcx** is the total number of boundary conditions in the x (or α) direction for the system. This value is inherited by any derived function of type **walsh**.
- The integer **nbcy** is the total number of boundary conditions in the y (or β) direction for the system. This value is inherited by any derived function of type **walsh**.
- The integer **nbcz** is the total number of boundary conditions in the z (or γ) direction for the system. This value is inherited by any derived function of type **walsh**.
- The integer **nbct** is the total number of boundary conditions in the t (or τ) direction for the system. This value is inherited by any derived function of type **walsh**.

- The logical flag **jacb** is true if the function is derived from (or is itself) a primary dependent variable or boundary condition. Often, PDEs include metric coefficients represented by Walsh functions across the domain that are only functions of the independent variables, in which case **jacb** is false. If a new function is derived from a Walsh function with **jacb** equal true, it too will assign **jacb** equal to true.

- The one-dimensional, integer array **nseg** of length 4 includes the number of segments across the domain in the x, y, z, and t directions (or α, β, γ, and τ directions), respectively. If a simulation includes only two directions then the segment numbers of the remaining directions are set to 1 - indicating a constant value of all functions in those directions. This array is inherited by any derived function of type **walsh**.

- The two-dimensional, real array **domain** with dimension (4,2) includes the initial, constant value of the computational coordinate in the x, y, z, and t directions (or α, β, γ, and τ directions), in the (1,1), (2,1), (3,1), and (4,1) locations, respectively. It includes the terminating, constant value of the computational coordinate in the x, y, z, and t directions (or α, β, γ, and τ directions), in the (1,2), (2,2), (3,2), and (4,2) locations, respectively. If a simulation includes only two directions then the initial and terminating values of the remaining directions are set to 0 and 1. This array is inherited by any derived function of type **walsh**.

- The one-dimensional, logical array **bdep** has length **ndep**. The mth element of **bdep** is true if it is derived from the mth primary, dependent variable. Otherwise it is false.

- The one-dimensional, logical array **bcx** has length **nbcx**. The mth element of **bcx** is true if it is derived from the mth boundary condition in the x (or α) direction. Otherwise it is false.

- The one-dimensional, logical array **bcy** has length **nbcy**. The mth element of **bcy** is true if it is derived from the mth boundary condition in the y (or β) direction. Otherwise it is false.

- The one-dimensional, logical array **bcz** has length **nbcz**. The mth element of **bcz** is true if it is derived from the mth boundary condition in the z (or γ) direction. Otherwise it is false.

- The one-dimensional, logical array **bct** has length **nbct**. The mth element of **bct** is true if it is derived from the mth boundary condition in the t (or τ) direction. Otherwise it is false.

- The one-dimensional, real array **comp** has length **len**. It contains the wave components A_l of the Walsh function series.

$$\sum_{n=1}^{\text{nseg}(4)} \sum_{k=1}^{\text{nseg}(3)} \sum_{j=1}^{\text{nseg}(2)} \sum_{i=1}^{\text{nseg}(1)} A_l g_i(x) g_j(y) g_k(z) g_n(t)$$

where $l = i + \text{nseg}(1)(j - 1 + \text{nseg}(2)(k - 1 + \text{nseg}(3)(n - 1)))$.

- The three-dimensional, real array **jac** is dimensioned (**len**, **len**, **ndep**). Jacobian **jac**(l, m, n) contains the partial derivative of the lth wave component with respect to the mth wave component of the nth primary, dependent variable.

- The three-dimensional, real array **jacbcx** is dimensioned to (**len**, lenx, max(1,**nbcx**)), where:

$$\text{lenx} = \max(1, \textbf{nseg(2)} * \textbf{nseg(3)} * \textbf{nseg(4)})$$

The element **jacbcx**(l,m,n) contains the partial derivative of the lth wave component with respect to the mth wave component of the nth boundary condition in the x (or α) direction.

- The three-dimensional, real array **jacbcy** is dimensioned to (**len**,leny, max(1,**nbcy**)), where:

$$\text{leny} = \max(1, \textbf{nseg(1) * nseg(3) * nseg(4)})$$

The element **jacbcy**(l,m,n) contains the partial derivative of the lth wave component with respect to the mth wave component of the nth boundary condition in the y (or β) direction.

- The three-dimensional, real array **jacbcz** is dimensioned to (**len**,lenz, max(1,**nbcz**)), where:

$$\text{lenz} = \max(1, \textbf{nseg(1) * nseg(2) * nseg(4)})$$

The element **jacbcz**(l,m,n) contains the partial derivative of the lth wave component with respect to the mth wave component of the nth boundary condition in the z (or γ) direction.

- The three-dimensional, real array **jacbct** is dimensioned to (**len**,lent, max(1,**nbct**)), where:

$$\text{lent} = \max(1, \textbf{nseg(1) * nseg(2) * nseg(3)})$$

The element **jacbct**(l,m,n) contains the partial derivative of the lth wave component with respect to the mth wave component of the nth boundary condition in the t (or τ) direction.

2.3 Operator Overloading

2.3.1 assignment(=)

```
public :: assignment(=)
interface assignment(=)
  module procedure walsh_assign_dd
end interface
```

A function of type **walsh** on the left side of the equals sign (=) is set equal to the result of the expression involving functions of type **walsh** on the right side.

2.3.2 operator(+)

```
public :: operator(+)
interface operator(+)
  module procedure walsh_plus_dd
  module procedure walsh_plus_dr
  module procedure walsh_plus_rd
end interface
```

A binary operation in which a function of type **walsh** may be added to another function of type **walsh** or to a real number.

2.3.3 operator(-)

```
public :: operator(-)
interface operator(-)
  module procedure walsh_minus_d
  module procedure walsh_minus_dd
  module procedure walsh_minus_dr
  module procedure walsh_minus_rd
end interface
```

A binary operation in which a function of type **walsh** may be subtracted from another function of type **walsh** or from a real number. The unary operation simply takes the negative of the function.

2.3.4 operator(*)

```
  public :: operator(*)
interface operator(*)
  module procedure walsh_prod_dd
  module procedure walsh_prod_rd
  module procedure walsh_prod_dr
end interface
```

A binary operation in which a function of type **walsh** may be multiplied by another function of type **walsh** or by a real number.

2.3.5 operator(**)

```
public :: operator(**)
interface operator(**)
  module procedure walsh_expo_di
end interface
```

A binary operation in which a function of type **walsh** may be raised to a positive integer power.

2.3.6 operator(/)

```
public :: operator(/)
interface operator(/)
  module procedure walsh_div_rd
  module procedure walsh_div_dr
  module procedure walsh_div_dd
end interface
```

If a function of type **walsh** occurs on the right side of the operator /, its reciprocal is evaluated using a Fast Walsh Reciprocal Transform and then the reciprocal multiplies the function or real number on the left. If a real number occurs on the right, its reciprocal multiplies the function on the left.

2.4 Functions

2.4.1 f_from_comp

```
pure elemental function f_from_comp(x, y, z, t, a, &
                                    filter)
  type(walsh),            intent(in) :: a
  real(dp),               intent(in) :: x, y, z, t
  logical,     optional,  intent(in) :: filter
  real(dp)                           :: f_from_comp
```

The value of a function **a** of type **walsh** at location (**x, y, z, t**) in the domain is returned as **f_from_comp**. If **filter** is present and true, the contributions from the highest family of wave numbers in each direction are not included in the function evaluation. This function does not utilize the Fast Walsh Transform (FWT) which may be more efficient in many circumstances.

2.4.2 absw

```
function absw(a)
  type(walsh), intent(in) :: a
  type(walsh)             :: absw
```

The absolute value of a function **a** of type **walsh** is returned. A Fast Walsh Transform (FWT) is used to convert from wave number component space to functional value space, the absolute value of the function is computed, and then another FWT transforms the result back to wave number component space.

2.4.3 sqrtw

```
function sqrtw(a)
  type(walsh), intent(in) :: a
  type(walsh)             :: sqrtw
```

The square root of a function **a** of type **walsh** is returned. A Fast Walsh Transform (FWT) is used to convert from wave number component space to functional value space, the square root of the function is computed, and then another FWT transforms the result back to wave number component space.

2.4.4 gn

```
pure elemental function gn(x0,x1,x,n)
  real(dp), intent(in)  :: x0
  real(dp), intent(in)  :: x1
  real(dp), intent(in)  :: x
  integer,  intent(in)  :: n
  real(dp)              :: gn
```

The value **gn** of the **n**th orthonormal walsh function, at location **x** in the domain, with lower bound **x0** and upper bound **x1**, is returned using Eq (4) of Reference [1]. In many cases, use of an FWT is more efficient.

2.4.5 pmap

```
pure elemental function pmap(n1,n2)
  integer, intent(in) :: n1, n2
  integer             :: pmap
```

The product of basis functions **n1** and **n2** map to **pmap**. That is: $g_{n1}(x)g_{n2}(x) = g_{pmap}(x)$ for domain length 1. See Table 2 of Reference [1].

2.4.6 intx

```
  pure function intx(f,fa,fb,diff)
    type(walsh),           intent(in) :: f
    type(walsh), optional, intent(in) :: fa
    type(walsh), optional, intent(in) :: fb
    logical,     optional, intent(in) :: diff
    type(walsh)                       :: intx
```

If **diff** is present and true, the returned function **intx** of type **walsh** defines the partial derivative of the input Walsh function **f** with respect to x (or coordinate α). If **diff** is absent or false, the returned function **intx** of type **walsh** defines the integral of the input Walsh function **f** as a function of x (or coordinate α). Evaluation of the derivative and integral differ primarily by the transformation matrix used: $\mathcal{D}$ for the derivative from Eq. 52 of Reference [1] or χ^T for the integral from Eq. 44 of Reference [1]. A Walsh function boundary condition variable, defining the variation of a boundary condition at $x = domain(1, 1)$ for **fa** or at $x = domain(1, 2)$ for **fb**, provides additional degrees of freedom to satisfy PDE system boundary conditions. Only one boundary condition (**fa** or **fb**) may be used in the function call.

2.4.7 inty

```
  pure function inty(f,fa,fb,diff)
    type(walsh),           intent(in) :: f
    type(walsh), optional, intent(in) :: fa
    type(walsh), optional, intent(in) :: fb
    logical,     optional, intent(in) :: diff
    type(walsh)                       :: inty
```

Same as **intx** but in the y (or coordinate β) direction.

2.4.8 intz

```
  pure function intz(f,fa,fb,diff)
    type(walsh),           intent(in) :: f
    type(walsh), optional, intent(in) :: fa
    type(walsh), optional, intent(in) :: fb
    logical,     optional, intent(in) :: diff
    type(walsh)                       :: intz
```

Same as **intx** but in the z (or coordinate γ) direction.

2.4.9 intt

```
  pure function intt(f,fa,fb,diff)
    type(walsh),           intent(in) :: f
    type(walsh), optional, intent(in) :: fa
    type(walsh), optional, intent(in) :: fb
    logical,     optional, intent(in) :: diff
    type(walsh)                       :: intt
```

Same as **intx** but in the t (or coordinate τ) direction.

2.5 Subroutines

2.5.1 allocate_walsh

```
pure subroutine allocate_walsh(a, nseg, ndep, nbcx,   &
                               nbcy, nbcz, nbct, jacb)
  type(walsh ),          intent(inout) :: a
  integer, dimension(4), intent(in)    :: nseg
  integer,               intent(in)    :: ndep
  integer,               intent(in)    :: nbcx
  integer,               intent(in)    :: nbcy
  integer,               intent(in)    :: nbcz
  integer,               intent(in)    :: nbct
  logical,               intent(in)    :: jacb
```

This routine allocates the elements of function **a** of derived type **walsh** and initializes its scalar elements. The allocatable vector elements of **a** are initialized to zero. If, on input, the allocatable elements of **a** are already allocated, the routine checks that the allocation is correct. If not correct, the elements are deallocated and then reallocated. The other inputs are defined previously for type(**walsh**). The domain boundaries **domain** are not changed by this routine.

2.5.2 deallocate_walsh

```
pure subroutine deallocate_walsh(a)
  type(walsh ),    intent(inout):: a
```

This routine deallocates all allocatable elements of **a**.

2.5.3 initialize_walsh

```
subroutine initialize_walsh(a, nseg, ndep, nbcx, nbcy,&
                     nbcz, nbct, domain, b, mseg, dep)
  type(walsh ),             intent(out):: a
  integer,  dimension(4),   intent(in) :: nseg
  integer,  dimension(4),   intent(in) :: mseg
  real(dp), dimension(4,2), intent(in) :: domain
  real(dp), dimension(:),   intent(in) :: b
```

```
  integer,                  intent(in) :: ndep
  integer,                  intent(in) :: nbcx
  integer,                  intent(in) :: nbcy
  integer,                  intent(in) :: nbcz
  integer,                  intent(in) :: nbct
  integer,      optional    intent(in) :: dep
```

Primary dependent variables of type **walsh** are initialized in this routine.

- **a**: The function of type **walsh** to be initialized.
- **nseg**, **ndep**, **nbcx**, **nbcy**, **nbcz**, **nbct**, **domain**: See definition in Sec. 2.2.2.
- **b**: A one-dimensional array of real values used to compute the initial wave components of **a** using a FWT. The **b** array may be dimensioned to either the total number of segments in the entire domain or to the number of segments on any boundary of the domain. If **b** is dimensioned according to the number of segments on a domain boundary, then that distribution is injected into the rest of the domain. For example, in a time-dependent problem, the known initial condition at $t = 0$ (domain(4,1)) is used to initialize the function at later times.
- **mseg**: The number of segments for each computational direction of the initializing array **b**. The array is ordered by $l = i + \mathbf{mseg}(\mathbf{1})(j - 1 + \mathbf{mseg}(\mathbf{2})(k - 1 + \mathbf{mseg}(\mathbf{3})(n - 1)))$.
- **dep**: An integer specifying the identity of the initialized variable used for defining the Jacobian. If **a** is the second of three primary dependent variables, then **ndep** = 3 and **dep** = 2. If **a** is the fourth of six boundary conditions in the x (or α) direction, then **ndep** = 6 and **dep** = 4. The initialization of every primary dependent variable will have identical values of **ndep**, **nbcx**, **nbcy**, **nbcz**, **nbct** that are determined by the system of PDEs being solved. The initialization of every boundary condition will have **nbcx** = **nbcy** = **nbcz** = **nbct** = 0, and **ndep** equal to the total number of boundary conditions in that direction on both the initial and terminal boundaries. If **dep** is absent or equal to 0, then the initialized function **a** is not dependent on any dependent variable or boundary condition.

2.5.4 fast_walsh_gather_4d

```
subroutine fast_walsh_gather_4d(a, b, need_jac,       &
                    direction, truncate)
  type(walsh ),             intent(in ) :: a
  type(walsh ),             intent(out) :: b
  logical,                  intent(in ) :: need_jac
  logical,                  intent(in ) :: direction
  integer,                  intent(in ) :: truncate
```

This routine is the single public interface for executing Fast Walsh Transforms (FWT). Other private routines exist internal to the WALSH_TOOLS module for executing parts of the transformation.

- **a**: Input function of type **walsh** to be transformed.

- **b**: Transformed output function of type **walsh**.
- **need_jac**: A logical flag indicating the need to transform the Jacobian elements of **a**. Transform the Jacobians if true.
- **direction**: A logical flag indicating the direction of the transformation. If true, then the transformation goes from discrete functional values at smallest segment centers in the domain to the Walsh function wave components. If false, the transformation goes from wave component space to discrete values.
- **truncate**: An integer flag indicating the level of truncation in the transformation.
  0 - no truncation
  1 - truncate highest family of terms in the first coordinate direction
  2 - truncate highest family of terms in the second coordinate direction
  3 - truncate highest family of terms in the third coordinate direction
  4 - truncate highest family of terms in the fourth coordinate direction
  5 - truncate highest family of terms in all coordinate directions

2.5.5 g_hat_bisection

```
subroutine g_hat_bisection(level_max)
  integer, intent(in) :: level_max
```

- **level_max**: The maximum value of p used in any of the computational coordinate directions where the maximum number of Walsh functions equals 2^p.

This routine calculates **gns** (see Sec. 2.1.2) by the original bisection algorithm of Reference[1]. This algorithm does not use recursion relations. It is less efficient than the recursion algorithm in subroutine gn_recursion) and it does not compute the multiplication table. It is retained to enable independent checks of the recursion algorithm.

2.5.6 g_recursion

```
subroutine g_recursion(level_max)
  integer, intent(in) :: level_max
```

- **level_max**: The maximum value of p used in any of the computational coordinate directions where the maximum number of Walsh functions equals 2^p.

This routine calculates **gns** (see Sec. 2.1.2) by the recursion algorithm of Reference[1].

2.5.7 check_walsh

```
subroutine check_walsh(a)
  type(walsh), intent(in) :: a
```

This routine prints information about a function **a** of type **walsh** which is often helpful in debugging new code. It prints all of the scalar and small array elements of the derived type and the first 8 elements of the larger arrays in the derived type.

2.5.8 setup_domain

```
subroutine setup_domain(sa,sb,p,overlap,N,ds,sap,sbp,L)
  real(dp), intent(in)  :: sa, sb   ! true endpoints
  integer,  intent(in)  :: p        ! N = 2**p
  integer,  intent(in)  :: overlap  ! 0 = ^1122,
                                    ! 1 = 1^122,
                                    ! 2 = 11^22
  integer,  intent(out) :: N        ! number of segments
  real(dp), intent(out) :: ds       ! segment length
  real(dp), intent(out) :: sap, sbp ! overlap endpoints
  real(dp), intent(out) :: L        ! sbp - sap
```

The solution of PDEs may employ overlap of neighboring domains or overlap of a single domain across a boundary. The effective boundary location is defined by Eq. 100 of Reference [1].

- **sa, sb**: The true endpoints of the domain to be solved.
- **p**: The number of Walsh functions in the direction defined by (**sa, sb**) is 2^p.
- **overlap**: An integer flag defining the extent of the overlap.
  0 - no overlap, the effective domain is exactly the true domain.
  1 - 1/2 segment overlap, the midpoint of the terminating, smallest segment is positioned over the true boundary.
  2 - full segment overlap, the terminating smallest segment is positioned across the true boundary.
- **N**: Maximum number of segments in the given computational direction.
- **ds**: Length of smallest segment in the effective domain.
- **sap, sbp**: Endpoints of the effective domain, including overlap.
- **L**: Length of the effective domain.

3 Boundary Module: BOUNDARY

The module BOUNDARY includes all functions and subroutines required to define Walsh function boundary conditions for the demonstration problems.

3.1 overlap

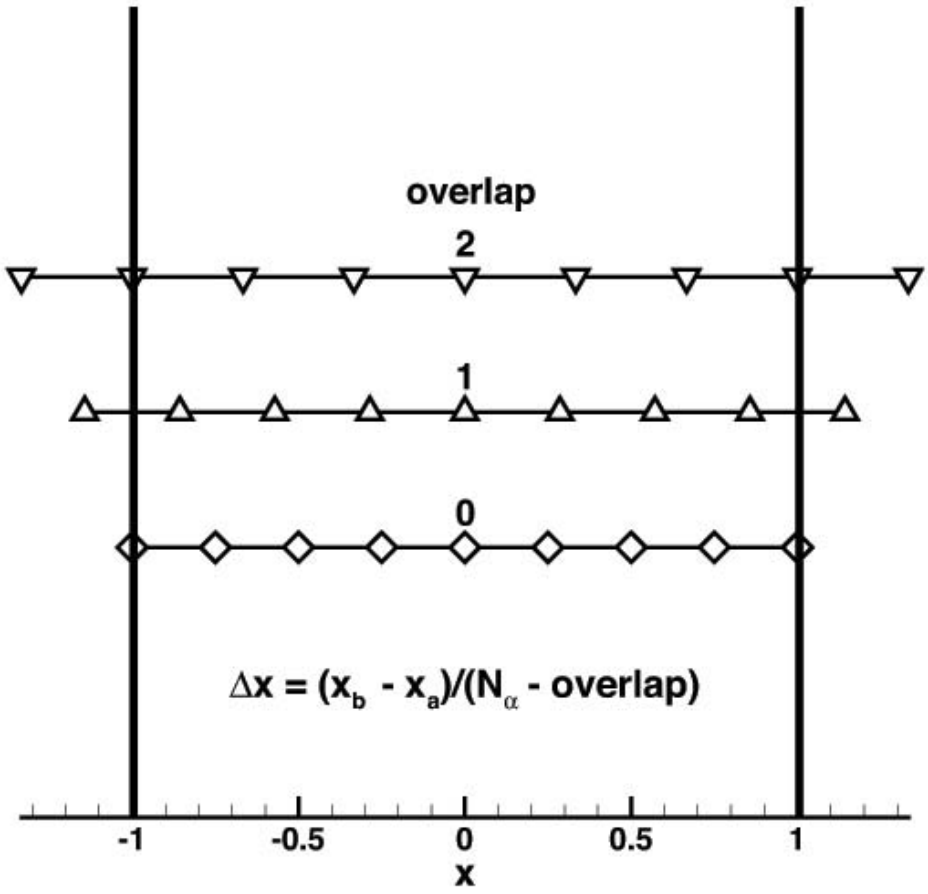


Figure 1: Schematic of the distribution of segments in $g_8(x)$ as a function of the **overlap** parameter.

The user-defined integer flag **overlap** is a key element of boundary condition formulation. As shown in Fig. 1, **overlap** defines the extent to which the terminating segments of the Walsh basis functions overlap the boundary of the physical domain. For **overlap** = 0, there is zero overlap. For **overlap** = 1, the midpoint of the terminating segment coincides with the boundary of the physical domain. The two half segments extending beyond the boundary sum to one. For **overlap** = 2, the terminating segments bound the physical domain. In this case, the two full segments

extending beyond the boundary sum to two. The total length of segments extending beyond the domain boundaries leads to the following equation for the smallest segment length in the Walsh series representation of the solution.

$$\Delta x = \frac{x_b - x_a}{N_\alpha - \mathbf{overlap}} \tag{1}$$

Here $N_\alpha = 2^{p_\alpha}$ is the terminating Walsh function index in the series representation,

$$q(x) = \sum_{\alpha=1}^{N_\alpha} \mathbf{qw}\%\mathbf{comp}(\alpha) g_\alpha(x) \tag{2}$$

and the symbol % indicates that the wave number component **comp** is an element of the Walsh function series type **qw** as defined in Sec. 2.2.2.

Numerical tests indicate that second-order differential equations are most accurately modeled with **overlap** =2. This overlap requires that the Walsh series representation of the dependent variable(s) has a value on the bounding segments equal to the given Dirichlet boundary conditions. A zero gradient boundary requires the values on the two terminating segments to be equal. Examples of the coding for these formulations are included in the demonstration test cases for Burgers Equation and the Riemann problem.

The first-order linear advection equation test case uses **overlap** = **1** to impose a periodic boundary condition. In this case, the dependent variable value on the left terminating segment at x_a is set equal to the value on the right terminating segment at x_b.

Figure 2 shows the distribution of segments across subdomains that span the global domain. Subdomains are introduced to reduce the storage requirements for Jacobians. In the example of Fig. 2, if a single series with 16 terms spans the global domain, then a single Jacobian with $16^2 = 256$ is evaluated. (Boundary condition contributions to the Jacobian are not included in this simple example.) If the global domain is subdivided into 4 equal parts (shown as black, green, red, and blue in Fig. 2), and each of these subdomains is spanned by a series with 4 terms, then the discretization of the global domain is equivalent to the single 16 term representation. However, the Jacobian storage is a factor of 4 smaller ($4 \times 4^2 = 64$) and Newton convergence is compromised when the global domain is partitioned in this manner.

When subdomains are used, inter-domain boundary conditions are required. The **overlap** parameter plays the same role in the inter-domain boundaries as in the global boundaries. For **overlap** = 0, neighboring subdomains (illustrated as black to green, green to red, red to blue in Fig. 2) have no overlap; their endpoints are coincident. For **overlap** =1, neighboring subdomains have one overlapping segment. For **overlap** = 2, neighboring subdomains have two overlapping segments. The three light black vertical lines in Fig. 2 define the inter-domain boundaries. The extent of overlap of the subdomains as a function of **overlap** is exactly the same as shown for the global domain boundaries in Fig. 1. The minimum segment size when n_D subdomains are utilized is given by

$$\Delta x = \frac{x_b - x_a}{n_D(N_\alpha - \mathbf{overlap})} \tag{3}$$

The advection test case uses **overlap** = 1 to compute inter-domain boundary conditions. The advection direction is from left to right. The function value computed at the far right segment of the black, green, and red sections is used to define the far left green, red, and blue segments, respectively. The Burgers equation and Riemann test cases include dissipation terms. In these

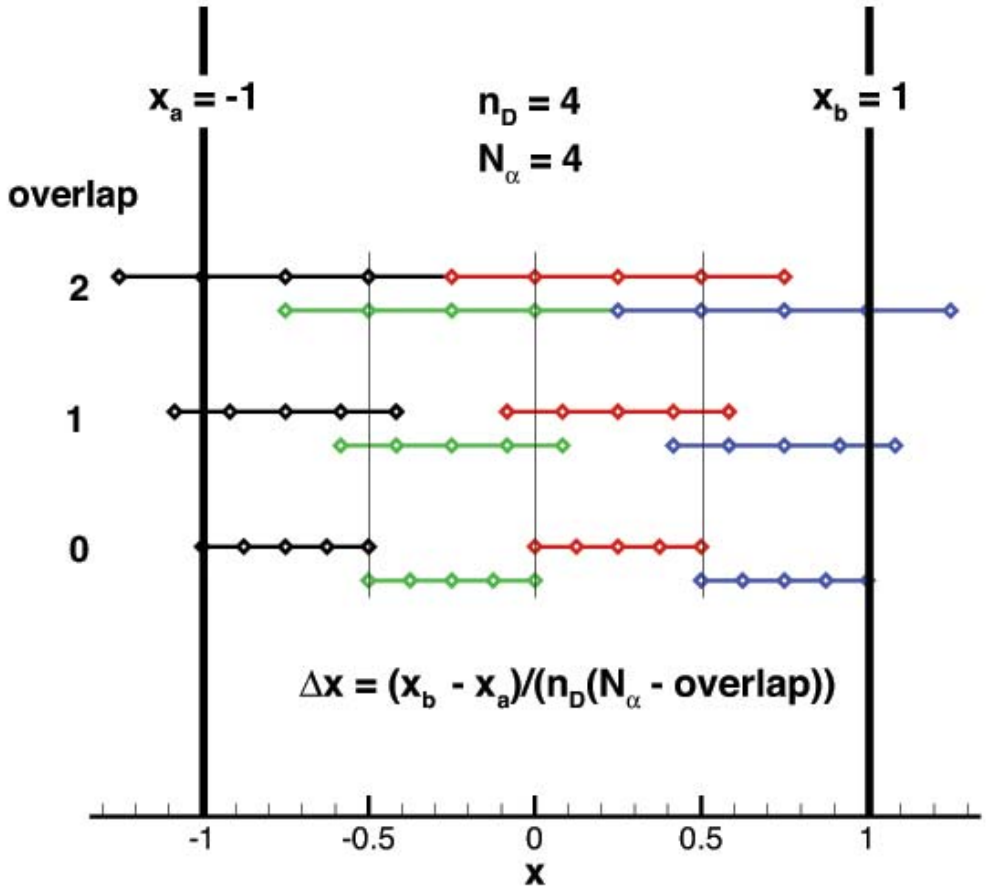


Figure 2: Schematic of the distribution of segments in the case where a four term Walsh series is implemented across each of four subdomains spanning the global domain.

cases, information is transferred in both directions across the inter-domain boundaries. The effect of this data transfer is to make the functional values at the overlapping segments of neighboring subdomains identical.

3.2 profile

```
function profile(xp,demo_code,var)
  real(dp), intent(in) :: xp
  integer,  intent(in) :: demo_code
  integer,  intent(in) :: var
  real(dp)             :: profile
```

The initial conditions for the three test problems are returned in **profile** as a function of **xp**.

- **xp**: position
- **demo_code**: test problem identification
  0 - advection problem
  1 - Burgers equation
  2 - Riemann problem
- **var**: returned primary dependent variable
  1 - density, ρ for **demo_code** = 2 , velocity, u for **demo_code** = 0 or 1
  2 - momentum, ρu for **demo_code** = 2
  3 - energy density, ρE for **demo_code** = 2

3.3 profile_burgers

```
function profile_burgers(x,nu)
  real(dp), intent(in) :: x, nu
  real(dp)             :: profile_burgers
```

The analytic, steady solution for Burgers equation is returned in **profile_burgers** computed as a function of **x** and **nu**.

3.4 profile_sod

```
subroutine profile_sod(x,t,p,rho,u,e)
  real(dp), intent(in ) :: x, t
  real(dp), intent(out) :: p,rho,u,e
```

The analytic, time dependent solution [3] for the Riemann problem is returned in **profile_sod** computed for the Sod test conditions [4].

- **x**: position, $-1 \leq x \leq 1$
- **t**: time
- **p**: pressure
- **rho**: density
- **u**: velocity
- **e**: energy

3.5 boundary_from_interior

```
public :: boundary_from_interior
interface boundary_from_interior
  module procedure  boundary_from_interior_1
  module procedure  boundary_from_interior_2
```

```
end interface
subroutine boundary_from_interior_1(qwp,qb,face,overlap,qbj)
  type(walsh),            intent(in)    :: qwp
  real(dp), dimension(:), intent(inout) :: qb
  integer,                intent(in)    :: face
                                        ! 1=xmin 2=xmax
                                        ! 3=ymin 4=ymax
                                        ! 5=zmin 6=zmax
                                        ! 7=tmin 8=tmax
  integer,                intent(in)    :: overlap
                                        ! 0=^1122
                                        ! 1=1^122
                                        ! 2=11^22
  real(dp), dimension(:,:), optional, intent(inout) :: qbj
!
subroutine boundary_from_interior_2(qwp,qb,face,overlap,qbj)
  type(walsh),            intent(in)    :: qwp
  real(dp), dimension(:), intent(inout) :: qb
  integer,                intent(in)    :: face
                                        ! 1=xmin 2=xmax
                                        ! 3=ymin 4=ymax
                                        ! 5=zmin 6=zmax
                                        ! 7=tmin 8=tmax
  integer,                intent(in)    :: overlap
                                        ! 0=^1122
                                        ! 1=1^122
                                        ! 2=11^22
  real(dp), dimension(:,:,:), optional, intent(inout) :: qbj
```

Boundary conditions are usually defined as a function of discrete values in physical space. Consequently, a FWT is used to transform components of primary dependent variables in sequency space to discrete values in physical space. A Walsh series dependent variable is denoted **qw**. The FWT of **qw** is denoted **qwp**. Both functions are of type **walsh**. The element **comp** of **qwp** is an array of discrete values at segment midpoints in the domain. The element **jac** of **qwp** is an array containing the Jacobian of discrete values of the dependent variable with respect to the wave components of the primary dependent variables.

- **qwp**: The FWT of **qw** in the domain.
- **qb**: The interpolated value of the discrete function **qwp%comp** on the boundary.
- **face**: Boundary code.
  1 = xmin, 2 = xmax
  3 = ymin, 4 = ymax
  5 = zmin, 6 = zmax
- **overlap**: An integer flag defining the extent of the overlap.
  0 - no overlap, the effective domain is exactly the true domain.

1 - 1/2 segment overlap, the midpoint of the terminating, smallest segment is positioned over the true boundary.
2 - full segment overlap, the terminating smallest segment is positioned across the true boundary

- **qbj**: The Jacobian of **qb** with respect to the wave components of **qw**. The argument is optional and if it is not present then a Jacobian does not need to be returned. In general, the interpolated value (**qb**) of the discrete function on the boundary is a function of **qwp** at two terminating segments for **overlap** =2. However, **qb** is a function of **qwp** at a single terminating segment for **overlap** = 0 or 1. If the entry is a two-dimensional, real array, then the Jacobian at the single terminating segment for **overlap** = 0 or 1 is returned. If the entry is a three-dimensional, real array, then the Jacobian at the terminating segment (third index =1) and its neighbor (third index = 2) for **overlap** = 2 is returned.

4 LAPACK Module: LAPACK

The linear solver **sgesv** from LAPACK [2] is required. That routine, and lower level routines required by **sgesv** have been assembled into a module to simplify installation. All real values used in these routines have been converted to real(dp) (see Sec. 2.1.1). Because the code is changed, the routine is renamed to **sgesv_mod** per request of the authors. [2]

5 Demonstration Program: DEMO

The demonstration program DEMO includes three test cases providing guidance on the use of the Walsh function module WALSH_TOOLS to solve partial differential equations. All of the sample problems are time dependent. Two are non-linear. One involves a system of equations.

If the temporal component is computed with $p_\tau = 0$, then the time derivative is approximated with a divided difference. That is $\frac{\partial q}{\partial t} \approx \frac{(q(x,t+\Delta t)-q(x,t))}{\Delta t}$. In this case of a single segment spanning time step **dt**, there are not enough degrees of freedom to use the differentiating matrix $\mathcal{D}$. [1] When $p_\tau > 0$, the differentiating matrix is employed and divided differences are not needed. Sample code blocks will be provided to illustrate this point.

5.1 Test Cases

5.1.1 Advection

The linear, first-order advection equation is

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{4}$$

where $c = 1$ is the wave speed and $u(x, 0) = u_0(x)$ is the initial condition on the domain $0 \le x \le 1$. The initial profile moves to the right. A periodic boundary condition is applied so that as the profile exits the right boundary it reemerges from the left. The test is designed to provide an easily evaluated metric to measure how well the initial profile is preserved. The initial profile in the demonstration case (see Fig. 3) is a sawtooth defined by

$$u_0(x) = \begin{cases} 0 & \text{for} \quad 0 \le x < \frac{1}{4} \\ 1 - 4|x - \frac{1}{2}| & \text{for} \quad \frac{1}{4} \le x < \frac{3}{4} \\ 1 & \text{for} \quad \frac{3}{4} \le x \le 1 \end{cases} \tag{5}$$

Code for Eq. 4 using WALSH_TOOLS is written:

```
if(N_tau > 1)then
  resw(1) = intt(q1w(m),fa=q10w(m),diff=.true.)                    &
          + wave_speed*intx(q1w(m),fa=q1aw(m),diff=.true.)
else
  resw(1) = (q1w(m) - q10w(m))/dt                                  &
          + wave_speed*intx(q1w(m),fa=q1aw(m),diff=.true.)
end if
```

In this example, if $N_\tau > 1$ ($p_\tau > 0$), then the time derivative term is calculated inside the function call to **intt** and it uses the differentiating matrix. If $N_\tau = 1$ ($p_\tau = 0$), then the time derivative term is calculated as a divided difference. In both cases, the space derivative is calculated inside the function call to **intx**. Note that for this linear, first-order equation, only a single Walsh function dependent variable **q1aw(m)** is required to satisfy a boundary condition in space and **q10w(m)** is required to satisfy an initial condition in time. Also note that index m refers to a subdomain. If $p_{domain} = 0$, then m is identically equal to 1.

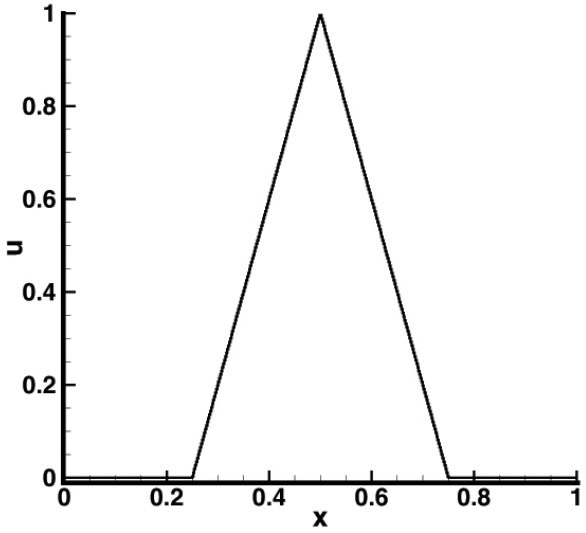


Figure 3: The initial profile for the advection test case, $u_0(x)$.

A subtle but critically important point is encountered when defining the dependent variable boundary condition **q10w(m)**. In general, one knows the initial condition. Why not simply represent **q10w(m)** explicitly as the Walsh transform of the known, initial condition? Different variations of this idea have been explored with no satisfactory solution. Numerical tests indicate that the computed **q10w(m)** provides a close but inexact representation of the initial condition on the segment center. Allowing **q10w(m)** to be part of the dependent variable set provides additional degrees of freedom that enables the computed solution on the global domain to exactly match the specified, initial conditions. This same observation applies to domain boundary conditions in space that are explicitly known. Treating boundary conditions as dependent variables provides additional relief at domain corners where spatial and temporal boundary conditions overlap. Close examination of the source code will reveal that boundary conditions at these corners are not overspecified; rather, the highest component Walsh function contribution to one of the boundary condition series is set to zero to reflect the fact that one less degree of freedom is required to close the system.

5.1.2 Burgers Equation

The non-linear, second-order Burgers equation is

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{6}$$

where the diffusivity $\nu \geq 0$ is input by the user. (See Sec. 5.3.) The initial condition is a linear function $u_0(x) = -x$ on the domain $-1 \leq x \leq 1$ with boundary conditions $u_a(t) = u(-1, t) = 1$ and $u_b(t) = u(1, t) = -1$.

Given these initial and boundary conditions, the solution profile evolves according to the local value of u. At any given time, points with positive value of u move to the right and points with

negative value of u move to the left with speed u. This motion is dissipated as a function of diffusivity ν. Large values of ν evolve profiles that are nearly linear between the boundary values. Small values of ν evolve profiles with an abrupt, shock-like transition from 1 to -1 at $x = 0$. Examples will be provided in Sec. 6.2.

If the diffusivity is specified 0, then an artificial diffusion term must be included to maintain a stable computation while accommodating boundary conditions from both the left and right. The artificial dissipation is defined

$$\nu_0 = \frac{1}{2}(dx_p)^2 \left| \frac{\partial u}{\partial x} \right| \tag{7}$$

where dx_p (given by Eq. 3) is the smallest segment size used in the Walsh series representation of the solution. The use of a second-order, artificial dissipation here and in the Riemann problem that follows presents a simple solution for communicating information from opposite boundaries. Such communication was not an issue in the linear advection test case in which all waves travel from left to right. It is thought that a characterstic-based formulation of this problem could offer better accuracy as $\nu \to 0$ and the transition from $u = 1$ to $u = -1$ occurs over a length smaller than dx_p. For now only the artificial dissipation is used to address this issue.

Code for Eq. 6 using WALSH_TOOLS is written:

```
if(N_tau > 1)then
  resw(1) = intt(q1w(m),fa=q10w(m),diff=.true.)                    &
          + intx(0.5_dp*q1w(m)**2-tauxw,fa=q1bw(m),diff=.true.)
else
  resw(1) = (q1w(m) - q10w(m))/dt                                  &
          + intx(0.5_dp*q1w(m)**2-tauxw,fa=q1bw(m),diff=.true.)
end if
```

Representation of the time derivative is exactly the same as discussed above for the advection equation. The shear term, **tauxw** $= \nu \partial u/\partial x$, was computed just above this block as:

```
uxw = intx(q1w(m),fa=q1aw(m),diff=.true.)
if(nu==0._dp)then
  dx2 = 0.5_dp*dx_eff**2
  tauxw = dx2*absw(uxw)*uxw
else
  tauxw = nu*uxw
end if
```

Note that if $\nu = 0$, then the artificial dissipation is added as defined in Eq. 7. Also note that this second-order equation has two boundary conditions that must be engaged. The Walsh function dependent variables **q1aw(m)** used in the definition of **uxw** $= \partial u/\partial x$ and **q1bw(m)** used in the definition of $\partial(.5u^2 - \nu \partial u/\partial x)/\partial x$, provide additional degrees of freedom required to satisfy the boundary conditions. The Walsh function dependent variable **q10w(m)** used in the definition of $\partial u/\partial t$ provides the additional degrees of freedom to satisfy the initial condition.

5.1.3 Riemann Problem

The Riemann problem in one-dimension defines the compressible gas dynamic flow that follows the breaking of a virtual diaphragm separating two, constant initial states. The solution typically

involves the propagation of a shock, a contact discontinuity, and an expansion fan. A particular instance of the Riemann problem was defined by Sod [4] and is frequently used in the validation of computational fluid dynamics codes.

The one-dimensional, time-dependent, compressible gas dynamic equations are:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} = 0 \tag{8}$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial (p + \rho u^2)}{\partial x} = 0 \tag{9}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u H}{\partial x} = 0 \tag{10}$$

$$p = (\gamma - 1)\rho e \tag{11}$$

$$e = E - \frac{u^2}{2} \tag{12}$$

$$H = E + \frac{p}{\rho} \tag{13}$$

The need for artificial dissipation was discussed in the previous section. It is added in Eqs 14 - 16.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left[\rho u - \nu_1 \frac{\partial \rho}{\partial x}\right] = 0 \tag{14}$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial}{\partial x}\left[p + \rho u^2 - \nu_2 \frac{\partial \rho u}{\partial x}\right] = 0 \tag{15}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial}{\partial x}\left[\rho u H - \nu_3 \frac{\partial \rho E}{\partial x}\right] = 0 \tag{16}$$

The artificial diffusion coefficients defined in Eqs. 17 - 19 are of order $(dx_p)^2$.

$$\nu_1 = (dx_p)^2 \left|\frac{\partial \rho}{\partial x}\right| + (dx_p)^2 \tag{17}$$

$$\nu_2 = (dx_p)^2 \left|\frac{\partial \rho u}{\partial x}\right| + (dx_p)^2 \tag{18}$$

$$\nu_3 = (dx_p)^2 \left|\frac{\partial \rho E}{\partial x}\right| + (dx_p)^2 \tag{19}$$

The equations are solved on the domain $-1 \le x \le 1$ with $\gamma = 1.4$. Constant initial conditions on the left ($x < 0$) and right ($x > 0$) for the Sod test case [4] are given by

$$p_L = 1 \qquad p_R = 0.1$$
$$\rho_L = 1 \qquad \rho_R = 0.125$$
$$u_L = 0 \qquad u_R = 0$$

Boundary conditions are given by

$$\frac{\partial \rho}{\partial x}(-1, t) = 0 \qquad \frac{\partial \rho}{\partial x}(1, t) = 0$$
$$\rho u(-1, t) = 0 \qquad \rho u(1, t) = 0$$
$$\frac{\partial \rho E}{\partial x}(-1, t) = 0 \qquad \frac{\partial \rho E}{\partial x}(1, t) = 0$$

Code for Eq. 14 - 16 using WALSH_TOOLS is written:

```
if(N_tau > 1)then
  resw(1) = intt(q1w(m),fa=q10w(m),diff=.true.)                    &
          + intx(q2w(m)-q1xw,fa=q1aw(m),diff=.true.)
  resw(2) = intt(q2w(m),fa=q20w(m),diff=.true.)                    &
          + intx(pw + rhou2w - q2xw,fa=q2aw(m),diff=.true.)
  resw(3) = intt(q3w(m),fa=q30w(m),diff=.true.)                    &
          + intx(q2w(m)*htw - q3xw,fa=q3aw(m),diff=.true.)
else
  resw(1) = (q1w(m) - q10w(m))/dt                                  &
          + intx(q2w(m)-q1xw,fa=q1aw(m),diff=.true.)
  resw(2) = (q2w(m) - q20w(m))/dt                                  &
          + intx(pw + rhou2w - q2xw ,fa=q2aw(m),diff=.true.)
  resw(3) = (q3w(m) - q30w(m))/dt                                  &
          + intx(q2w(m)*htw - q3xw,fa=q3aw(m),diff=.true.)
end if
```

The main difference between this code sample and previous ones in this section is that there are three dependent variables and three corresponding residual equations to be solved.

5.2 Compilation

Any modern FORTRAN compiler should be satisfactory for creating the executable. In the present example, the FORTRAN compiler "gfortran" is used. Create the following four files in a working directory and enter the command "make" to create the executable "demo".

5.2.1 Makefile.env

```
SHELL = /bin/sh
F90FLAGS = -O2 -g
F90COMPILER = gfortran
F90LINKER   = gfortran
F90LIBS = -lm
```

5.2.2 make.dependencies

```
demo.o: demo.f90 \
 walsh_tools.o \
 lapack.o \
 boundary.o
walsh_tools.o: walsh_tools.f90
lapack.o: lapack.f90 \
 walsh_tools.o
boundary.o: boundary.f90 \
 walsh_tools.o
```

5.2.3 Makefile.rules

Note that the lines with eight leading spaces is a "tab" key entry.

```
FLAGSFILE=.compileFlags
## object file rules
.SUFFIXES :
.SUFFIXES : .c .F90 .f90 .o
.f90.o:
        $(F90COMPILER) -c $(F90FLAGS) $*.f90
.F90.o:
        @echo "$(PKGFLAGS)" > $(FLAGSFILE)
        $(F90COMPILER) -c $(F90FLAGS) $(PKGFLAGS) $*.F90
.c.o:
        $(CCOMPILER) -c $(CFLAGS) $(PKGFLAGS) $(PKGINCLUDE) $*.c
```

5.2.4 Makefile

Note that the lines with eight leading spaces is a "tab" key entry.

```
include Makefile.env
include Makefile.rules
TARGET=demo
default:
        $(MAKE) $(TARGET)
$(TARGET): demo.o
        $(F90LINKER) $(LDFLAGS) -o $@ *.o $(F90LIBS)
clean:
        -rm -rf *.o *.stb *.mod $(TARGET)
include make.dependencies
```

5.3 Inputs

Upon entering the command "demo" the following prompts are provided.

5.3.1 demo_code

```
Enter code for demo. 0=Advection, 1=Burgers, 2=Riemann
```

The answer to this prompt is stored as variable **demo_code** and it controls the selection of the demonstration problem.

5.3.2 nu

If **demo_code** = 1 the next prompt is

```
Enter value for diffusivity (0 for inviscid)
```

A value of $\nu \geq 0$ is required. If $0 < \nu < (dx_p)^2$, then the expected shock transition thickness is less than the smallest segment size and the solution may not converge. In this case, the artificial diffusivity ν_0 as defined in Eq. 7 is engaged by specifying $\nu = 0$.

5.3.3 p_alpha

The next prompt is

```
Enter power of 2 for series g_alpha(x)
```

An integer value for p_α is required. The number of terms in the Walsh series solution in the x direction is $N_\alpha = 2^{p_\alpha}$.

5.3.4 p_tau

The next prompt is

```
Enter power of 2 for series g_tau(t)
```

An integer value for p_τ is required. The number of terms in the Walsh series solution in the t direction is $N_\tau = 2^{p_\tau}$.

5.3.5 p_domain

The next prompt is

```
Enter power of 2 for number of domains spanning x
```

An integer value for p_{domain} is required. The number of subdomains across the x direction is $n_D = 2^{p_{domain}}$ as discussed in Sec. 3.1. If $p_{domain} = 0$ the subdomain is identical to the global domain.

5.3.6 overlap_x

The next prompt is

```
Enter code for overlap of x-domains: 0=^1122, 1=1^122, 2=11^22
```

An integer value for **overlap_x** is required. Guidance for selection of integer code 0, 1, or 2 has been provided in Sec. 3.1.

5.3.7 overlap_t

The next prompt is

```
Enter code for overlap of t-domains: 0=^1122, 1=1^122, 2=11^22
```

An integer value for **overlap_t** is required. Guidance for selection of integer code 0, 1, or 2 has been provided in Sec. 3.1.

5.3.8 dt

The next prompt is

```
Enter timestep
```

A real value for the time step **dt** is required. This entry defines the domain size in t.

5.3.9 t_max

The next prompt is

```
Enter total time
```

A real value for the total time to be simulated **t_max** is required. The solution is advanced in time with time step **dt** until total time **t_max** is attained.

5.3.10 truncate

The next prompt is

```
Enter truncate: 1=yes, 0 = no
```

An integer value for the truncation flag **truncate** is required. Truncation, if selected, truncates the contribution of the highest family of Walsh functions after each time step, damping solution oscillations in the vicinity of shocks.

5.4 OutputFiles

5.4.1 Screen Output

The error norm (Eq. 20) is printed as a function of time. The norm is computed relative to the time accurate solution for the Advection equation and the Riemann problem. It is computed relative to the steady state solution ($t >> 0$) for Burgers equation. Note that variable names with subscript "e" represent exact solutions to the test problems.

$$\text{error norm} = \begin{cases} \sum_{\alpha=1}^{N_\alpha} |u_e(x_\alpha, t_\tau) - u(x_\alpha, t_\tau)| dx_p & \text{for the Advection test case} \\ \\ \sum_{\alpha=1}^{N_\alpha} |u_e(x_\alpha, \infty) - u(x_\alpha, t_\tau)| dx_p & \text{for the Burgers test case} \\ \\ \sum_{\alpha=1}^{N_\alpha} [|\rho_e(x_\alpha, t_\tau) - \rho(x_\alpha, t_\tau)|/\rho_e(x_\alpha, t_\tau) & \\ \qquad +|p_e(x_\alpha, t_\tau) - p(x_\alpha, t_\tau)|/p_e(x_\alpha, t_\tau) & \\ \qquad +|u_e(x_\alpha, t_\tau) - u(x_\alpha, t_\tau)|]\, dx_p & \text{for the Riemann test case} \end{cases} \tag{20}$$

The L1 norm metric for convergence of wave component magnitudes and integration constants is printed as **l1norm** for every Newton relaxation step. It is the sum of the absolute value of the change in these dependent variables for each relaxation step divided by the total number of dependent variables.

5.4.2 Plot Files

The solution is output for plotting using ASCII format Tecplot [5] files. A constant over segment (COS) format is used in some files to emphasize the Walsh function provenance of the solution. This format produces figures that appear as a stair step distribution where the width of the stairs equals dx_p. Other files use a segment centered (SC) format in which the exact solution at the segment center is compared to the Walsh function solution at the segment center.

- advection.dat: x, u in COS format for the advection test case.
- advection_exact.dat: x, u_e, u in SC format for the advection test case.
- burgers.dat: x, u in COS format for the Burgers Equation test case.
- burgers_exact.dat: x, u_e, u in SC format for the Burgers Equation test case.
- tube.dat: x, ρ, u, e, p in COS format for the Riemann problem.
- tube_exact.dat: x, ρ_e, ρ, u_e, u, e_e, e, p_e, p in SC format for the Riemann problem.

A separate Tecplot zone is defined for each time step. Animations of the solution as a function of time can be created by looping over zones. In the case of the Riemann problem, if the number of segments in the domain is greater than 32, then only every tenth time step solution is saved in order to reduce file size. Note that the error norm returns a value of zero for time steps that are not saved.

5.5 Flowchart

The logical flow through program DEMO is designed to enable easy contrast of function and subroutine calls for each of the three test problems. The logical flow through the code is relatively simple. Comments in the body of the code correspond to the following list which highlight key steps in the algorithm. Differences between the three demonstration code cases in each section should provide better understanding regarding the way that WALSH_TOOLS is used.

- Begin input and setup Parameters as function of overlap
- Open files for tecplot
- Begin allocate and define initial conditions for each dependent variable
- Determine maximum exponent of 2 and compute Walsh functions accordingly
- Determine dimensions of working arrays, allocate and initialize dependent var
- Start marching in time
- Capture initial condition from previous time step
- Initialize loop for Newton iterations
- Define residuals for PDEs in the interior
- Load residuals and Jacobians from the interior for linear solve
- Gather data from interior to define dependent variables on the boundaries
- Define dependent variables and their Jacobians on left and right boundaries
- Define dependent variables and their Jacobians on initial boundaries
- Compute residuals for boundaries

- Solve (drdq) dq = res
- Update dependent variables
- Converged or maximum number of allowed relaxations for this time step - Record solution for post-processing
- Proceed to next time step if elapsed time $<$ t_max

6 Sample Solutions

6.1 Advection

The first advection test case uses the following input:

```
0      ! demo_code
8      ! p_alpha
2      ! p_tau
0      ! p_domain
1      ! overlap_x
1      ! overlap_t
.01    ! dt
1.     ! t_max
0      ! truncate
```

The screen output associated with the first two time steps is:

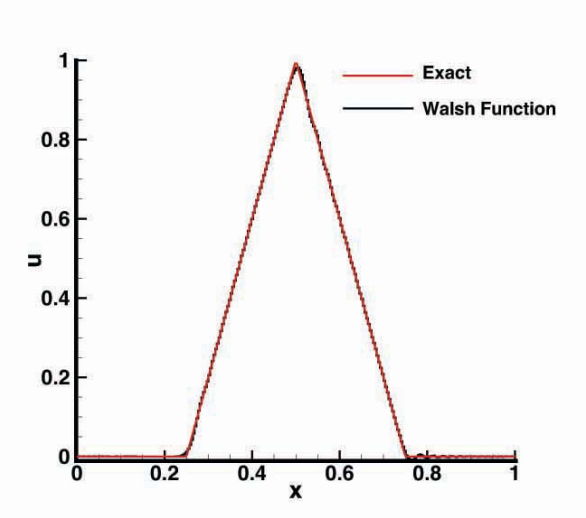
```
 After            1  global relaxation steps, l1norm =    5.6617743259134005E-005
 After            2  global relaxation steps, l1norm =    4.3534691685466348E-018
 At time =    1.0000000000000000E-002  error norm =    4.8274819263081860E-005
 After            1  global relaxation steps, l1norm =    1.0070258277040343E-004
 After            2  global relaxation steps, l1norm =    4.7764211618465652E-018
 At time =    2.0000000000000000E-002  error norm =    6.2407283720140061E-005
```

The convergence criteria requires **l1norm** $< 10^{-10}$ before advancing to the next time. After 100 time steps, completing a single cycle of the initial sawtooth profile, the output associated with the last two time steps is:

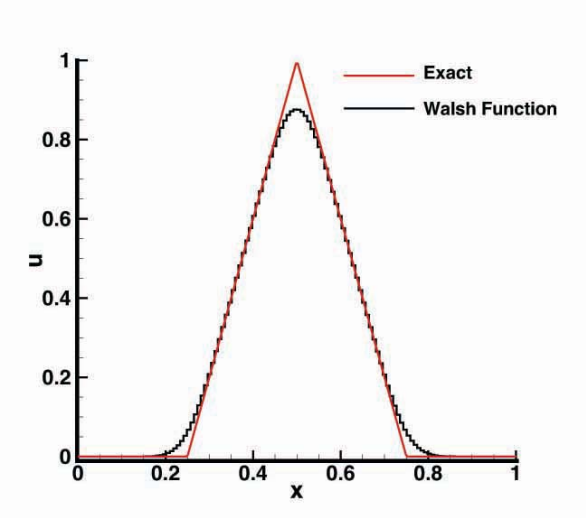
```
 At time =   0.98000000000000065       error norm =    1.6334071728664030E-003
 After            1  global relaxation steps, l1norm =    1.2065793876161089E-004
 After            2  global relaxation steps, l1norm =    4.4603672023197547E-018
 At time =   0.99000000000000066       error norm =    1.6609375712891763E-003
 After            1  global relaxation steps, l1norm =    9.7874071929656260E-005
 After            2  global relaxation steps, l1norm =    4.0288670095828081E-018
 At time =    1.0000000000000007       error norm =    1.6595651815910024E-003
```

The Walsh function solution of the advected profile after one cycle is compared to the exact solution in Fig. 4a. If a Courant number is defined as the ratio of the distance traveled by a wave in time step **dt** divided by the smallest segment size dx_p, then the Courant number in this case is $c\mathrm{d}t/dx_p = 2.55$. The error norm after one cycle equals 1.66 10^{-3}. Some oscillation is evident at the front foot of the profile. If the case is rerun with **truncate** = 1 to smooth oscillations, the error norm after 1 cycle increases to 1.37 10^{-2} and the profile is shown in Fig. 4b. Truncation eliminates the oscillations at the expense of smoothing out the discontinuities at the tip and feet of the profile.

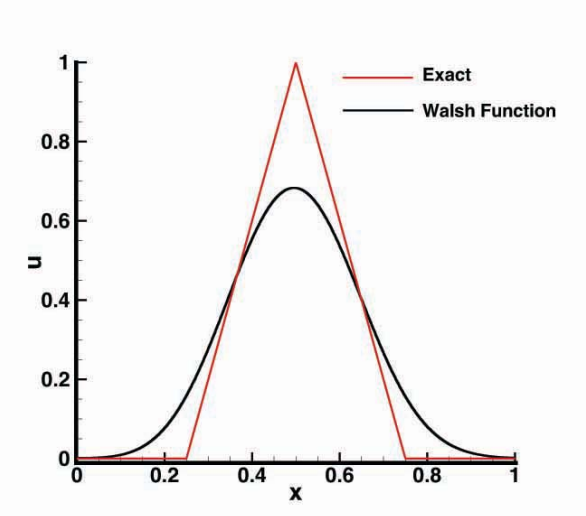
If temporal degrees of freedom are exchanged for spatial degrees of freedom by decreasing p_τ to 0 and increasing p_α to 10, then the Courant number increases to 10.23 and the profile appears in Fig. 4c. Note that $p_\tau = 0$ requires **overlap_t** = 0. The temporal truncation error overwhelms any benefit from a factor 4 finer resolution in x in this case.



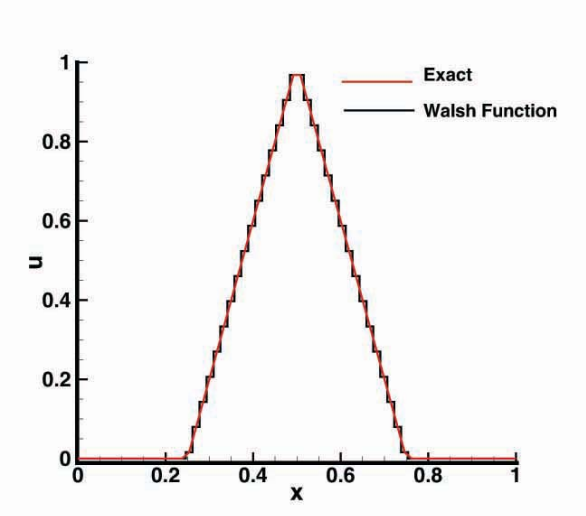
(a) $p_\alpha = 8$, $p_\tau = 2$, truncate = 0, dt = 0.01



(b) $p_\alpha = 8$, $p_\tau = 2$, truncate = 1, dt = 0.01



(c) $p_\alpha = 10$, $p_\tau = 0$, truncate = 0, dt = 0.01



(d) $p_\alpha = 6$, $p_\tau = 6$, truncate = 0, dt = 1.

Figure 4: Advection test problem showing profile after one complete cycle at $t = 1$. Black is the Walsh function series solution. Red is the exact solution.

In stark contrast to the previous case, if spatial resolution is sacrificed for increased temporal revolution by setting $p_\alpha = 6$ and $p_\tau = 6$ and increasing the time step **dt** by a factor of 100, then a complete cycle is computed in a single time step as shown in Fig. 4d with zero error norm for the linear problem. Zero error means that the advected profile exactly crosses the segment midpoints. The Courant number in this case is 63, using the previous definition. A more representative definition of the Courant number is to consider the distance traveled by a wave in the smallest temporal segment size $c(dt)_p$ divided by the smallest segment size in space $(dx)_p$, in which case the Courant number is equal to 1. This result exhibits a resonance in that the computed profile exactly repeats

every time step, so that the **l1norm** = 0 in a single relaxation step, because the initial condition at the beginning of the time step equals the converged condition at the end of the time step. This convergence behavior is captured in the following screen output for this case.

```
 After            1  global relaxation steps, l1norm =    9.0004947907115784E-004
 After            2  global relaxation steps, l1norm =    2.8991809556234448E-018
 At time =    1.0000000000000000       error norm =    6.3363534456109310E-017
 After            1  global relaxation steps, l1norm =    2.7318494984685376E-018
 At time =    2.0000000000000000       error norm =    1.7762526354977382E-016
 After            1  global relaxation steps, l1norm =    2.4981190159567315E-018
 At time =    3.0000000000000000       error norm =    3.2152011681236220E-016
```

6.2 Burgers Equation

The first advection test case uses the following input:

```
1       ! demo_code
.1      ! nu
6       ! p_alpha
0       ! p_tau
0       ! p_domain
2       ! overlap_x
0       ! overlap_t
0.1     ! dt
10.     ! t_max
0       ! truncate
```

The screen output associated with the first two time steps is:

```
 After            1  global relaxation steps, l1norm =    1.3058307122319951E-002
 After            2  global relaxation steps, l1norm =    7.5467963857695639E-010
 After            3  global relaxation steps, l1norm =    6.3699131433843720E-018
 At time =   0.10000000000000001       error norm =   0.63941814680754439
 After            1  global relaxation steps, l1norm =    6.2273326570300310E-004
 After            2  global relaxation steps, l1norm =    4.6192751541354147E-010
 After            3  global relaxation steps, l1norm =    2.3647276144270429E-018
 At time =   0.20000000000000001       error norm =   0.56102491910795882
```

The screen output associated with the last two time steps is:

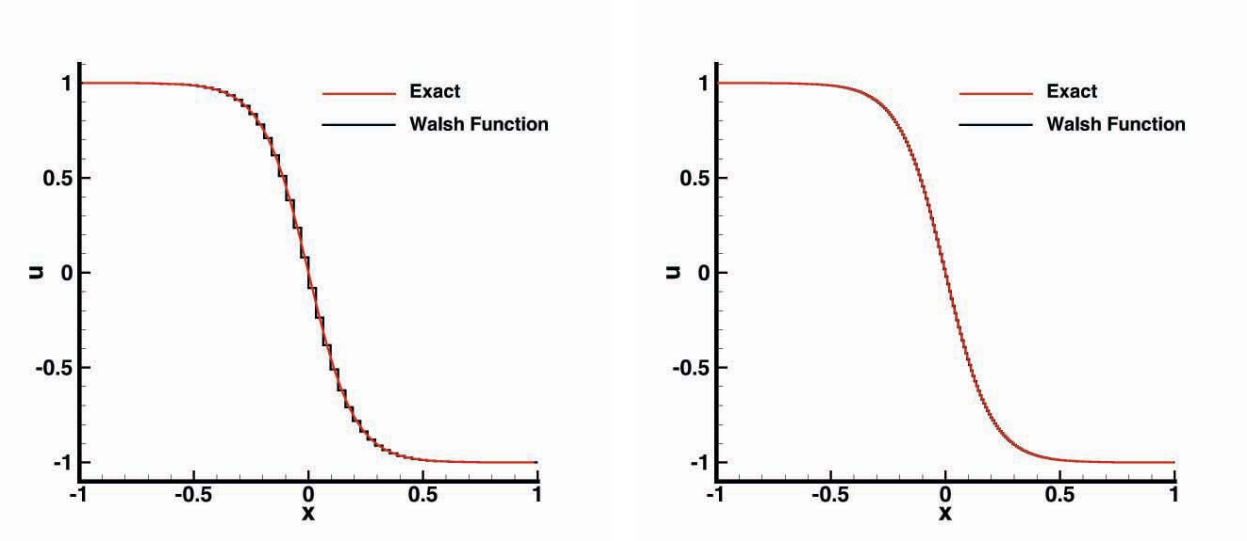
```
  After            1  global relaxation steps, l1norm =    9.6066902157564443E-014
 At time =    9.9999999999999805       error norm =    5.8941163924331765E-004
 After            1  global relaxation steps, l1norm =    7.4717043270173795E-014
 At time =    10.099999999999980       error norm =    5.8941163870657035E-004
```

A steady solution is obtained for $t > 6.6$ based on attaining an **l1norm** $< 10^{-10}$ in a single relaxation step following an advance in time. The steady solution at $t = 10$ is shown in Fig. 5a. The segmented nature of the underlying Walsh function support is evident in the stair stepping

Table 1: Error norms for Burgers Equation with $\nu = 0.1$.

p_α	error norm	error norm ratio
3	$1.28\ 10^{-1}$	-
4	$1.33\ 10^{-2}$	9.62
5	$2.60\ 10^{-3}$	5.12
6	$5.89\ 10^{-4}$	4.41
7	$1.40\ 10^{-4}$	4.21
8	$3.45\ 10^{-5}$	4.06
9	$8.92\ 10^{-6}$	3.87

appearance of the solution. The stair stepping is less evident in Fig. 5b in which $p_\alpha = 8$ and segment size is a factor of 4 smaller. The error norm for this problem is recorded in Table 1. With each increment in p_α the number of segments is doubled and the error norm decreases by a factor of approximately 4, indicating second-order accuracy relative to the smallest segment size.

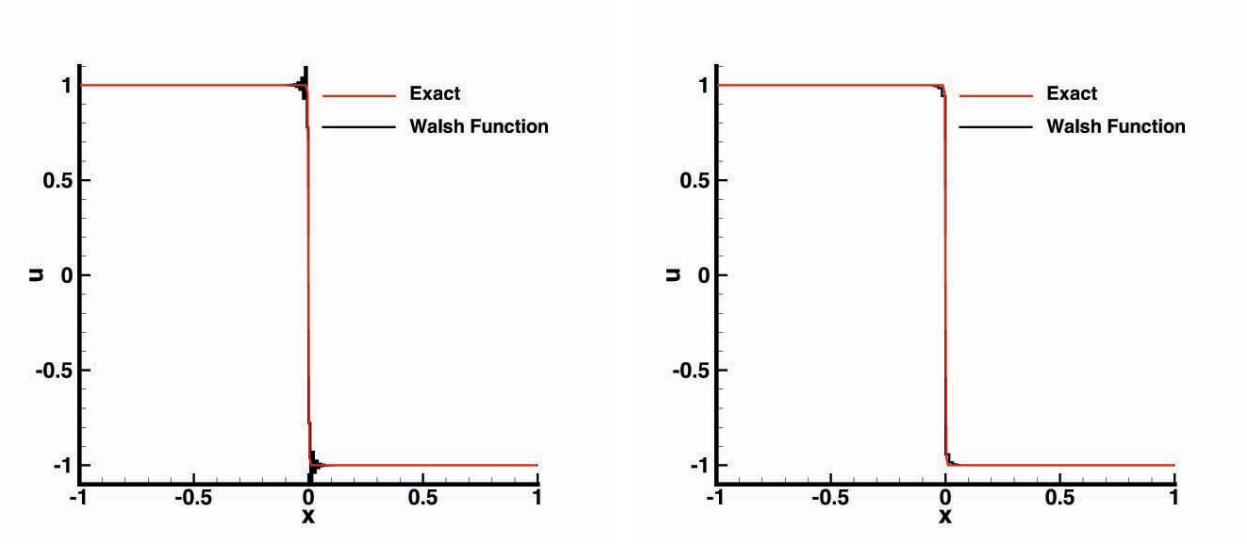


(a) $p_\alpha = 6$ (b) $p_\alpha = 8$

Figure 5: Effect of p_α in case of $\nu = 0.1$, $p_\tau = 0$, and $p_{domain} = 0$ for Burgers Equation. Black is the Walsh function series solution. Red is the exact solution.

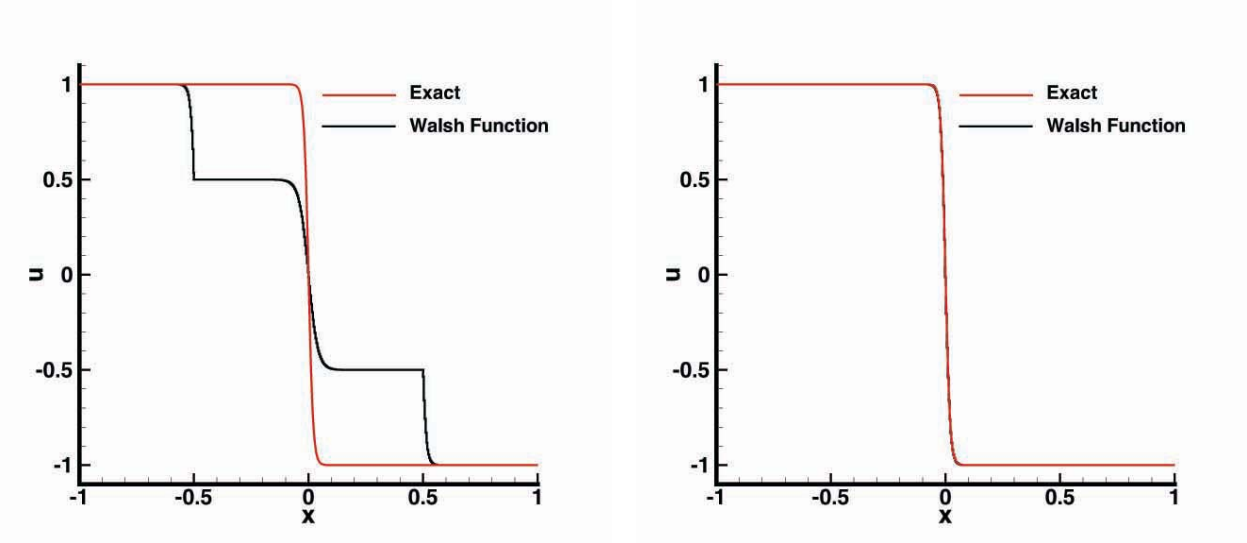
As the shock thickness decreases below the width of $(dx)_p$ with decreasing ν, oscillations may be encountered in the solution. These oscillations are eliminated by truncating the highest family of Walsh function contributions to the solution at the conclusion of a time step. An example is presented in Fig. 6, where the solution without (a) and with (b) truncation is shown when $\nu = 0.001$ and $p_\alpha = 8$. The error norm without truncation is $7.27\ 10^{-3}$ and with truncation is $1.94\ 10^{-3}$, indicating roughly a factor of 4 decrease in error. It is a subtle but important point to

note that the highest order Walsh functions still contribute to the lower-order non-linear solution retained after truncation through the self-mapping property under multiplication.



(a) **truncate** = 0 (b) **truncate** = 1

Figure 6: Effect of **truncate** in case of $\nu = 0.001$, $p_\alpha = 8$, $p_\tau = 0$, and $p_{domain} = 0$ for Burgers Equation. Black is the Walsh function series solution. Red is the exact solution.



(a) **overlap** = **1** (b) **overlap** = **2**

Figure 7: Effect of **overlap** in case of multiple subdomains with $\nu = 0.01$, $p_\alpha = 8$, $p_\tau = 0$, and $p_{domain} = 2$. Black is the Walsh function series solution. Red is the exact solution.

The effect of **overlap** in the context of a simulation with multiple subdomains is illustrated in Fig. 7 for $p_\alpha = 8$, $p_\tau = 0$, and $p_{domain} = 2$. The interdomain boundary condition must preserve both the value of u and $\partial u/\partial x$ across the boundary. The **overlap_x** $= 1$ environment preserves only the value of u at the single, shared terminating segments. The solution slope is left free to abruptly rotate across the shared boundary. The **overlap_x** $= 2$ environment preserves the value of u at the two terminating segments of neighboring subdomains. By pinning two values across the boundary, the solution slopes are also forced to agree. Consequently, with **overlap_x** $= 1$, the solution at the subdomain boundaries essentially stays frozen at the inititial condition of the left boundary for waves moving to the right ($u > 0$), and stays frozen at the initial condition of the right boundary for waves moving to the left ($u < 0$). The initial condition was a linear variation such that $u(x, 0) = u_0(x) = -x$. (See Sec. 5.1.2.)

6.3 Riemann Problem

The Riemann test case uses the following input:

```
2  ! demo_code
7  ! p_alpha
0  ! p_tau
3  ! p_domain
2  ! overlap_x
0  ! overlap_t
0.001 ! dt
2. ! t_max
0  ! truncate
```

The screen output for the first time step is:

```
 After            1  global relaxation steps, l1norm =    2.8007668595445945E-003
 After            2  global relaxation steps, l1norm =    3.2658025447789479E-004
 After            3  global relaxation steps, l1norm =    1.4103432439592479E-004
 After            4  global relaxation steps, l1norm =    1.6225866477887173E-005
 After            5  global relaxation steps, l1norm =    5.4659371674958619E-006
 After            6  global relaxation steps, l1norm =    1.0297314400298979E-006
 After            7  global relaxation steps, l1norm =    2.3522205149719938E-007
 After            8  global relaxation steps, l1norm =    5.2232478176224048E-008
 After            9  global relaxation steps, l1norm =    1.3932483790545194E-008
 After           10  global relaxation steps, l1norm =    1.8865180918122111E-009
 After           11  global relaxation steps, l1norm =    4.3475882498435346E-010
 After           12  global relaxation steps, l1norm =    1.7182955939989943E-010
 After           13  global relaxation steps, l1norm =    1.0161033064457939E-010
 After           14  global relaxation steps, l1norm =    6.8963576126975439E-011
```

The solution for density at $t = 0.42$ is presented in Fig. 8. The expansion moving to the left is evident for $-.5 < x < 0$. The contact discontinuity at $x \approx 0.35$ and the shock at $x \approx 0.7$ move to the right. The constant states between the shock and the contact discontinuity, and between the contact discontinuity and the head of the expansion, are in excellent agreement with the exact

solution. The tail of the expansion at $x \approx -0.5$ is slightly rounded. The contact discontinuity appears more dissipated than the shock. The largest differences are thought to be associated with the direction of characteristics approaching the discontinuities.

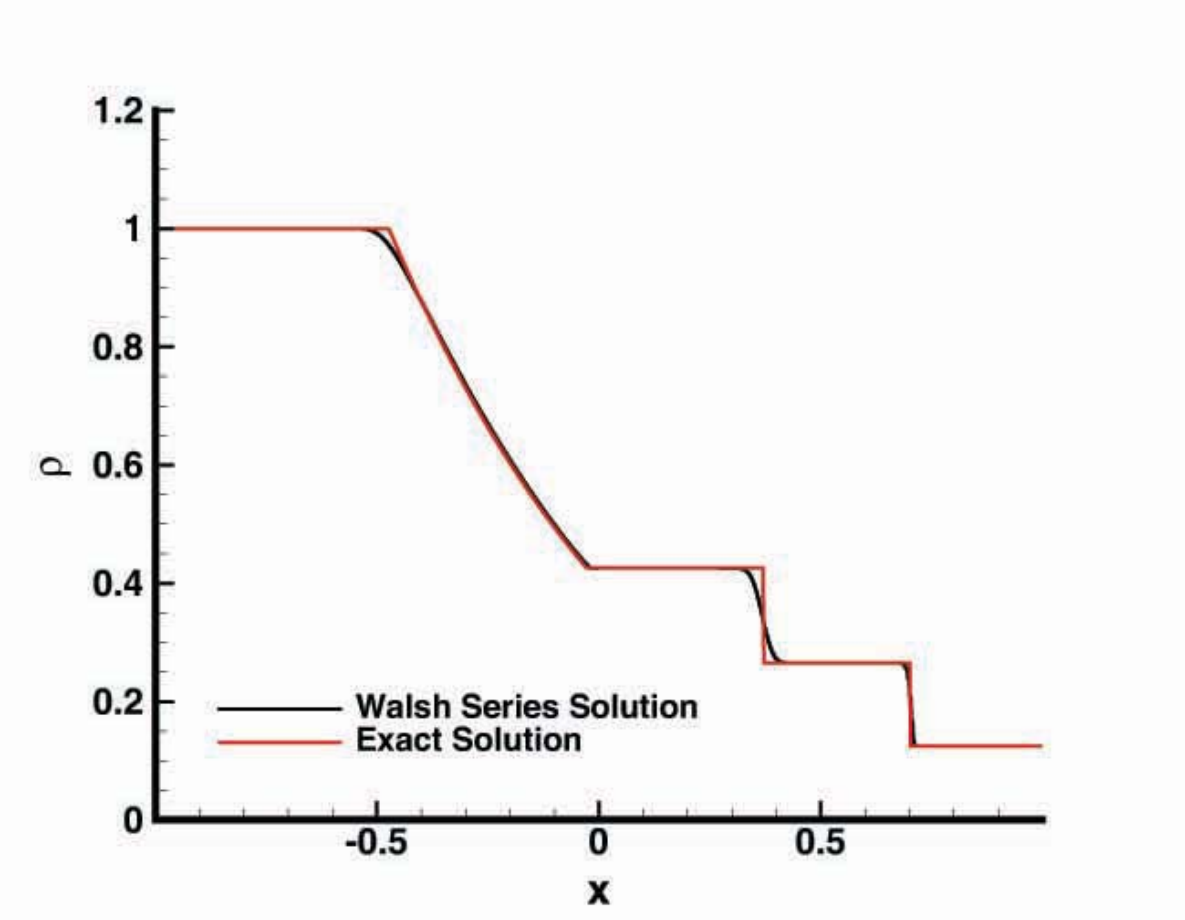


Figure 8: Density profile at $t = 0.42$ for Riemann problem. Black is the Walsh function series solution. Red is the exact solution.

7 Summary

The module WALSH_TOOLS has been developed and documented here to enable new users to explore and advance orthonormal Walsh function analyses of nonlinear, partial differential equations. The module contains a fundamental set of operators, functions, and subroutines that enable a new user to operate with Walsh functions in much the same way that Fourier analysis tools are currently utilized. The major difference of the Walsh function approach for nonlinear problems is the property of closure under multiplication - the product of any two Walsh functions is exactly another Walsh function.

Three test problems documented herein include linear advection, Burgers Equation, and a Riemann problem using Sod test conditions. The first case provides examples of a profile with discontinuities in slope that is advected across the domain. The other two cases explore shock capturing and ability to accurately track a moving discontinuity. Code for these options appear together in the program so that new users may study the source code, along with narrative provided here, to better understand how the infrastructure in WALSH_TOOLS may be used in new applications. Of special note is that the infrastructure automatically provides Jacobians needed to obtain Newton relaxation of all equations and boundary conditions across the domain.

The algorithms described herein should be considered a starting point for applying Walsh function based simulations of nonlinear, partial differential equations. Improvements are expected and will be reported as they become available.

References

1. Gnoffo, P. A.: Global Series Solutions of Nonlinear Differential Equations with Shocks Using Walsh Functions. *J. Comput. Phys.*, Vol. 258, Feb 2014, pp. 650–688.

2. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; and Sorensen, D.: *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third ed., 1999.

3. F. D. Lora-Clavijo, J. P. Cruz-Perez, F. Siddhartha Guzman, and J. A. Gonzalez: Exact solution of the 1D riemann problem in Newtonian and relativistic hydrodynamics. *Revista Mexicana de Fisica E*, Vol. 59, 2013, pp. 28–50.

4. Sod, G. A.: A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *J. Comput. Phys.*, Vol. 27, 1978, pp. 1–31.

5. *tecplot.360 2013 User's Manual: Release 1*. Bellevue, Washington, 2013.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 10 - 2014	Technical Memorandum	

4. TITLE AND SUBTITLE

A Walsh Function Module Users' Manual

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Gnoffo, Peter A.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

470883.04.07.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-20399

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/TM-2014-218536

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified
Subject Category 61
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The solution of partial differential equations (PDEs) with Walsh functions offers new opportunities to simulate many challenging problems in mathematical physics. The approach was developed to better simulate hypersonic flows with shocks on unstructured grids. It is unique in that integrals and derivatives are computed using simple matrix multiplication of series representations of functions without the need for divided differences. The product of any two Walsh functions is another Walsh function - a feature that radically changes an algorithm for solving PDEs. A FORTRAN module for supporting Walsh function simulations is documented. A FORTRAN code is also documented with options for solving time-dependent problems: an advection equation, a Burgers equation, and a Riemann problem. The sample problems demonstrate the usage of the Walsh function module including such features as operator overloading, Fast Walsh Transforms in multi-dimensions, and a Fast Walsh reciprocal.

15. SUBJECT TERMS

Burgers; Closure; Riemann; Shock capturing; Sod

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	47	(443) 757-5802