Users' Manual

Version 3.0.0
August 28, 2020

BRAID

PARALLEL MULTIGRID IN TIME

Center for Applied Scientific Computing (CASC), LLNL
Department of Mathematics and Statistics, University of New Mexico

Contents

1 Abstract

This package implements an optimal-scaling multigrid solver for the (non)linear systems that arise from the discretization of problems with evolutionary behavior. Typically, solution algorithms for evolution equations are based on a time-marching approach, solving sequentially for one time step after the other. Parallelism in these traditional time-integration techniques is limited to spatial parallelism. However, current trends in computer architectures are leading towards systems with more, but not faster, processors, i.e., clock speeds are stagnate. Therefore, faster overall runtimes must come from greater parallelism. One approach to achieve parallelism in time is with multigrid, but extending classical multigrid methods for elliptic operators to this setting is a significant achievement. In this software, we implement a non-intrusive, optimal-scaling time-parallel method based on multigrid reduction techniques. The examples in the package demonstrate optimality of our multigrid-reduction-in-time algorithm (MGRIT) for solving a variety of equations in two and three spatial dimensions. These examples can also be used to show that MGRIT can achieve significant speedup in comparison to sequential time marching on modern architectures.

It is **strongly recommended** that you also read `Parallel Time Integration with Multigrid` after reading the Overview of the XBraid Algorithm. It is a more in depth discussion of the algorithm and associated experiments.

2 XBraid Quickstart, User Advice, and License

What is XBraid?

XBraid is a parallel-in-time software package. It implements an optimal-scaling multigrid solver for the (non)linear systems that arise from the discretization of problems with evolutionary behavior.

This code and associated algorithms are developed at Lawrence Livermore National Laboratory, and at collaborating academic institutions.

For our publication list, please go here. There you will papers on XBraid and various application areas where XBraid has been applied, e.g., fluid dynamics, machine learning, parabolic equations, Burgers' equation, powergrid systems, etc.

About XBraid

Typically, solution algorithms for evolution equations are based on a time-marching approach, solving sequentially for one time step after the other. Parallelism in these traditional time-integration techniques is limited to spatial parallelism. However, current trends in computer architectures are leading towards systems with more, but not faster, processors, i.e., clock speeds are stagnate. Therefore, faster overall runtimes must come from greater parallelism. Our approach to achieve such parallelism in time is with multigrid.

In this software, we implement a non-intrusive, optimal-scaling time-parallel method based on multigrid reduction techniques (multigrid-reduction-in-time or MGRIT). A few important points about XBraid are as follows.

- The algorithm enables a scalable parallel-in-time approach by applying multigrid to the time dimension.
- It is designed to be nonintrusive. That is, users apply their existing sequential time-stepping code according to our interface, and then XBraid does the rest. Users have spent years, sometimes decades, developing the right time-stepping scheme for their problem. XBraid allows users to keep their schemes, but enjoy parallelism in the time dimension.
- XBraid solves exactly the same problem that the existing sequential time-stepping scheme does.
- XBraid is flexible, allowing for a variety of time stepping, relaxation, and temporal and spatial coarsening options.
- The full approximation scheme multigrid approach is used to accommodate nonlinear problems.
- XBraid written in MPI/C with C++, Fortran 90, and Python interfaces.
- XBraid is released under LGPL 2.1.

Documentation

- For examples of using XBraid, see the `examples/` and `drivers/` directories, and in particular examples/ex-01-$*$
- See the release page for links to precompiled documentation PDFs that go through, step-by-step, how to use XBraid.
- For tutorials, see the bottom of our publications page.
- For citing XBraid, see here.

Advice to Users

The field of parallel-in-time methods is in many ways under development, and success has been shown primarily for problems with some parabolic character. While there are ongoing projects (here and elsewhere) looking at varied applications such as hyperbolic problems, computational fluid dynamics, power grids, medical applications, and so on, expectations should take this fact into account. That being said, we strongly encourage new users to try our code for their application. Every new application has its own issues to address and this will help us to improve both the algorithm and the software. Please see our project publications website for our recent `publications` concerning some of these varied applications.

For bug reporting, please use the issue tracker here on Github. Please include as much relevant information as possible, including all the information in the “VERSION” file located in the bottom most XBraid directory. For compile and runtime problems, please also include the machine type, operating system, MPI implementation, compiler, and any error messages produced.

Building XBraid

- To specify the compilers, flags and options for your machine, edit makefile.inc. For now, we keep it simple and avoid using configure or cmake.
- To make the library, libbraid.a,

  ```
  $ make
  ```

- To make the examples

  ```
  $ make all
  ```

- The makefile lets you pass some parameters like debug with

  ```
  $ make debug=yes
  ```

  or

  ```
  $ make all debug=yes
  ```

  It would also be easy to add additional parameters, e.g., to compile with insure.

- To set compilers and library locations, look in makefile.inc where you can set up an option for your machine to define simple stuff like

  ```
  CC = mpicc
  MPICC = mpicc
  MPICXX = mpiCC
  LFLAGS = -lm
  ```

Meaning of the name

We chose the package name XBraid to stand for Time-Braid, where X is the first letter in the Greek word for time, Chronos. The algorithm braids together time-grids of different granularity in order to create a multigrid method and achieve parallelism in the time dimension.

License

3 Introduction

3.1 Overview of the XBraid Algorithm

The goal of XBraid is to solve a problem faster than a traditional time marching algorithm. Instead of sequential time marching, XBraid solves the problem iteratively by simultaneously updating a space-time solution guess over all time values. The initial solution guess can be anything, even a random function over space-time. The iterative updates to the solution guess are done by constructing a hierarchy of temporal grids, where the finest grid contains all of the time values for the simulation. Each subsequent grid is a coarser grid with fewer time values. The coarsest grid has a trivial number of time steps and can be quickly solved exactly. The effect is that solutions to the time marching problem on the coarser (i.e., cheaper) grids can be used to correct the original finest grid solution. Analogous to spatial multigrid, the coarse grid correction only *corrects* and *accelerates* convergence to the finest grid solution. The coarse grid does not need to represent an accurate time discretization in its own right. Thus, a problem with many time steps (thousands, tens of thousands or more) can be solved with 10 or 15 XBraid iterations, and the overall time to solution can be greatly sped up. However, this is achieved at the cost of more computational resources.

To understand how XBraid differs from traditional time marching, consider the simple linear advection equation, $u_t = -cu_x$. The next figure depicts how one would typically evolve a solution here with sequential time stepping. The initial condition is a wave, and this wave propagates sequentially across space as time increases.

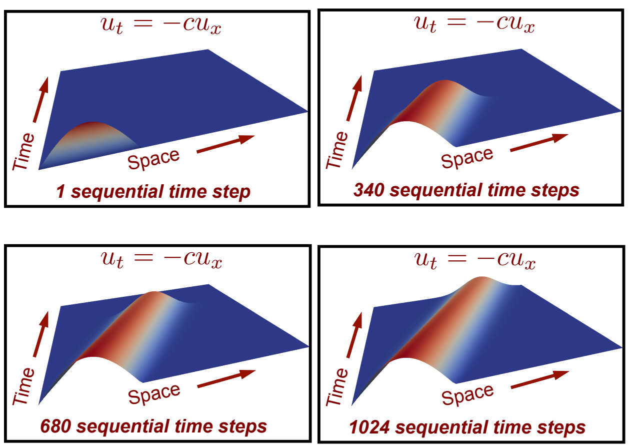


Figure 1 Sequential time stepping.

XBraid instead begins with a solution guess over all of space-time, which for demonstration, we let be random. An XBraid iteration does

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values. Relaxation is just a local application of the time stepping scheme, e.g., backward Euler.
2. Restriction to the first coarse grid, i.e., interpolate the problem to a grid that contains fewer time values, say every second or every third time value.
3. Relaxation on the first coarse grid
4. Restriction to the second coarse grid and so on...
5. When a coarse grid of trivial size (say 2 time steps) is reached, it is solved exactly.
6. The solution is then interpolated from the coarsest grid to the finest grid

One XBraid iteration is called a *cycle* and these cycles continue until the solution is accurate enough. This is depicted in the next figure, where only a few iterations are required for this simple problem.

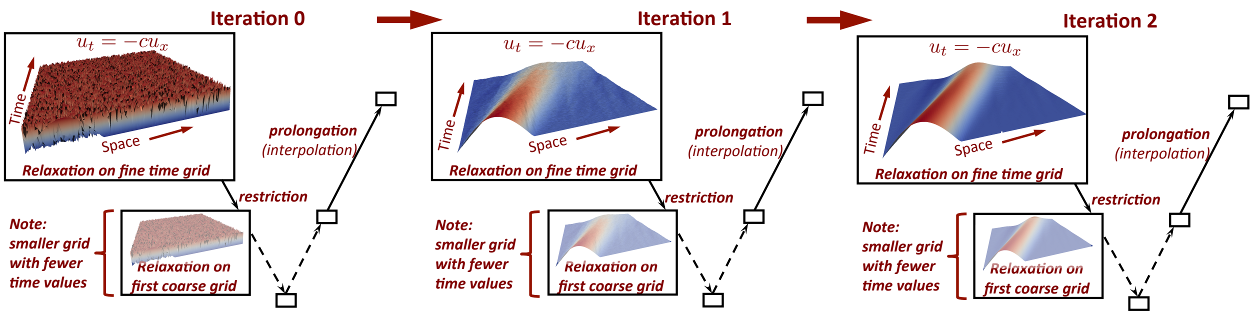


Figure 2 XBraid iterations.

There are a few important points to make.

- The coarse time grids allow for global propagation of information across space-time with only one XBraid iteration. This is visible in the above figure by observing how the solution is updated from iteration 0 to iteration 1.
- Using coarser (cheaper) grids to correct the fine grid is analogous to spatial multigrid.
- Only a few XBraid iterations are required to find the solution over 1024 time steps. Therefore if enough processors are available to parallelize XBraid, we can see a speedup over traditional time stepping (more on this later).
- This is a simple example, with evenly space time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes.

To firm up our understanding, let's do a little math. Assume that you have a general system of ordinary differential equations (ODEs),

$$\boldsymbol{u}'(t) = \boldsymbol{f}(t, \boldsymbol{u}(t)), \quad \boldsymbol{u}(0) = \boldsymbol{u}_0, \quad t \in [0, T].$$

Next, let $t_i = i\delta t, i = 0, 1, ..., N$ be a temporal mesh with spacing $\delta t = T/N$, and u_i be an approximation to $u(t_i)$. A general one-step time discretization is now given by

$$\begin{aligned}\boldsymbol{u}_0 &= g_0\\ \boldsymbol{u}_i &= \Phi_i(\boldsymbol{u}_{i-1}) + \boldsymbol{g}_i, \quad i = 1, 2, ..., N.\end{aligned}$$

Traditional time marching would first solve for $i = 1$, then solve for $i = 2$, and so on. For linear time propagators $\{\Phi_i\}$, this can also be expressed as applying a direct solver (a forward solve) to the following system:

$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_0 \\ \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_N \end{pmatrix} = \begin{pmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \vdots \\ \boldsymbol{g}_N \end{pmatrix} \equiv \mathbf{g}$$

or

$$A\mathbf{u} = \mathbf{g}.$$

This process is optimal and O(N), but it is sequential. XBraid achieves parallelism in time by replacing this sequential solve with an optimal multigrid reduction iterative method [1] applied to only the time dimension. This approach is

- nonintrusive, in that it coarsens only in time and the user defines Φ. Thus, users can continue using existing time stepping codes by wrapping them into our framework.
- optimal and O(N), but O(N) with a higher constant than time stepping. Thus with enough computational resources, XBraid will outperform sequential time stepping.

[1] Ries, Manfred, Ulrich Trottenberg, and Gerd Winter. "A note on MGR methods." Linear Algebra and its Applications 49 (1983): 1-26.

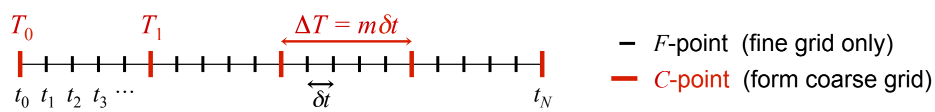
- highly parallel

We now describe the two-grid process in more detail, with the multilevel analogue being a recursive application of the process. We also assume that Φ is constant for notational simplicity. XBraid coarsens in the time dimension with factor $m > 1$ to yield a coarse time grid with $N_\Delta = N/m$ points and time step $\Delta T = m\delta t$. The corresponding coarse grid problem,

$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix},$$

is obtained by defining coarse grid propagators $\{\Phi_\Delta\}$ which are at least as cheap to apply as the fine scale propagators $\{\Phi\}$. The matrix A_Δ has fewer rows and columns than A, e.g., if we are coarsening in time by 2, A_Δ has one half as many rows and columns.

This coarse time grid induces a partition of the fine grid into C-points (associated with coarse grid points) and F-points, as visualized next. C-points exist on both the fine and coarse time grid, but F-points exist only on the fine time scale.



Every multigrid algorithm requires a relaxation method and an approach to transfer values between grids. Our relaxation scheme alternates between so-called F-relaxation and C-relaxation as illustrated next. F-relaxation updates the F-point values $\{u_j\}$ on interval (T_i, T_{i+1}) by simply propagating the C-point value u_{mi} across the interval using the time propagator $\{\Phi\}$. While this is a sequential process, each F-point interval update is independent from the others and can be computed in parallel. Similarly, C-relaxation updates the C-point value u_{mi} based on the F-point value u_{mi-1} and these updates can also be computed in parallel. This approach to relaxation can be thought of as line relaxation in space in that the residual is set to 0 for an entire time step.

The F updates are done simultaneously in parallel, as depicted next.

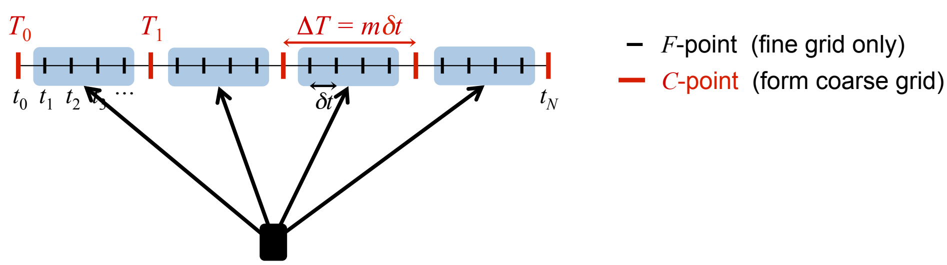


Figure 3 Update all F-point intervals in parallel, using the time propagator Φ.

Following the F sweep, the C updates are also done simultaneously in parallel, as depicted next.

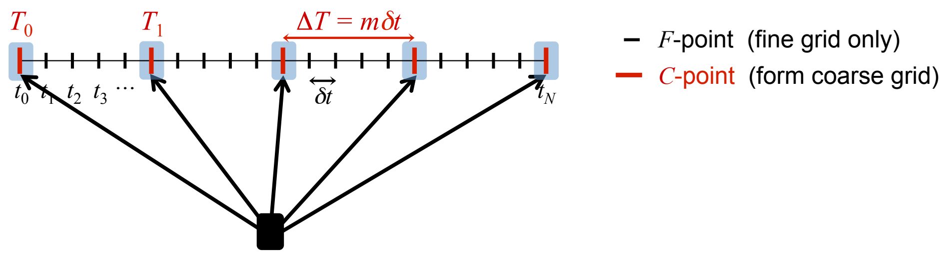


Figure 4 Update all C-points in parallel, using the time propagator Φ.

In general, FCF- and F-relaxation will refer to the relaxation methods used in XBraid. We can say

- FCF- or F-relaxation is highly parallel.
- But, a sequential component exists equaling the number of F-points between two C-points.
- XBraid uses regular coarsening factors, i.e., the spacing of C-points happens every m points.

After relaxation, comes forming the coarse grid error correction. To move quantities to the coarse grid, we use the restriction operator R which simply injects values at C-points from the fine grid to the coarse grid,

$$R = \begin{pmatrix} I & & \\ 0 & & \\ \vdots & & \\ 0 & & \\ & I & \\ & 0 & \\ & \vdots & \\ & 0 & \\ & & \ddots \end{pmatrix}^T.$$

The spacing between each I is $m-1$ block rows. While injection is simple, XBraid always does an F-relaxation sweep before the application of R, which is equivalent to using the transpose of harmonic interpolation for restriction (see `Parallel Time Integration with Multigrid`). Another interpretation is that the F-relaxation compresses the residual into the C-points, i.e., the residual at all F-points after an F-relaxation is 0. Thus, it makes sense for restriction to be injection.

To define the coarse grid equations, we apply the Full Approximation Scheme (FAS) method, which is a nonlinear version of multigrid. This is to accommodate the general case where f is a nonlinear function. In FAS, the solution guess and residual (i.e., $\mathbf{u}, \mathbf{g} - A\mathbf{u}$) are restricted. This is in contrast to linear multigrid which typically restricts only the residual equation to the coarse grid. This algorithmic change allows for the solution of general nonlinear problems. For more details, see this `PDF` by Van Henson for a good introduction to FAS. However, FAS was originally invented by Achi Brandt.

A central question in applying FAS is how to form the coarse grid matrix $A_\triangle$, which in turn asks how to define the coarse grid time stepper $\Phi_\triangle$. One of the simplest choices (and one frequently used in practice) is to let $\Phi_\triangle$ simply be Φ but with the coarse time step size $\Delta T = m\delta t$. For example, if $\Phi = (I - \delta t A)^{-1}$ for some backward Euler scheme, then $\Phi_\triangle = (I - m\delta t A)^{-1}$ would be one choice.

With this $\Phi_\triangle$ and letting $\mathbf{u}_\triangle$ be the restricted fine grid solution and $\mathbf{r}_\triangle$ be the restricted fine grid residual, the coarse grid equation

$$A_\triangle(\mathbf{v}_\triangle) = A_\triangle(\mathbf{u}_\triangle) + \mathbf{r}_\triangle$$

is then solved. Finally, FAS defines a coarse grid error approximation $\mathbf{e}_\triangle = \mathbf{v}_\triangle - \mathbf{u}_\triangle$, which is interpolated with P_Φ back to the fine grid and added to the current solution guess. Interpolation is equivalent to injecting the coarse grid to the

C-points on the fine grid, followed by an F-relaxation sweep (i.e., it is equivalent to harmonic interpolation, as mentioned above about restriction). That is,

$$P_\Phi = \begin{pmatrix} I & & \\ \Phi & & \\ \Phi^2 & & \\ \vdots & & \\ \Phi^{m-1} & & \\ & I & \\ & \Phi & \\ & \Phi^2 & \\ & \vdots & \\ & \Phi^{m-1} & \\ & & \ddots \end{pmatrix},$$

where m is the coarsening factor. See Two-Grid Algorithm for a concise description of the FAS algorithm for MGRIT.

3.1.1 Two-Grid Algorithm

The two-grid FAS process is captured with this algorithm. Using a recursive coarse grid solve (i.e., step 3 becomes a recursive call) makes the process multilevel. Halting is done based on a residual tolerance. If the operator is linear, this FAS cycle is equivalent to standard linear multigrid. Note that we represent A as a function below, whereas the above notation was simplified for the linear case.

1. Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation
2. Restrict the fine grid approximation and its residual:

   $$\mathbf{u}_\Delta \leftarrow R\mathbf{u}, \quad \mathbf{r}_\Delta \leftarrow R(\mathbf{g} - A(\mathbf{u}),$$

   which is equivalent to updating each individual time step according to

   $$u_{\Delta,i} \leftarrow u_{mi}, \quad r_{\Delta,i} \leftarrow g_{mi} - A(\mathbf{u})_{mi} \quad \text{for} \quad i = 0, ..., N_\Delta.$$

3. Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$
4. Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$
5. Correct: $\mathbf{u} \leftarrow \mathbf{u} + P\mathbf{e}_\Delta$
   This is equivalent to updating each individual time step by adding the error to the values of $\mathbf{u}$ at the C-points:

   $$u_{mi} = u_{mi} + e_{\Delta,i},$$

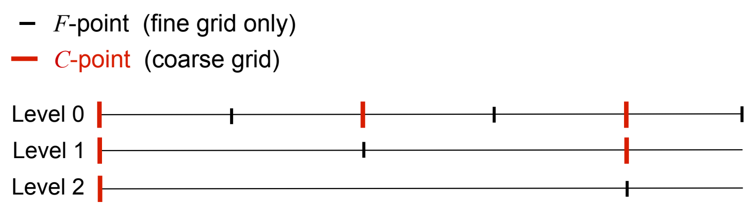
   followed by an F-relaxation sweep applied to $\mathbf{u}$.

3.1.2 Summary

In summary, a few points are

- XBraid is an iterative solver for the global space-time problem.
- The user defines the time stepping routine Φ and can wrap existing code to accomplish this.
- XBraid convergence will depend heavily on how well Φ_Δ approximates Φ^m, that is how well a time step size of $m\delta t = \Delta T$ will approximate m applications of the same time integrator for a time step size of δt. This is a subject of research, but this approximation need not capture fine scale behavior, which is instead captured by relaxation on the fine grid.

- The coarsest grid is solved exactly, i.e., sequentially, which can be a bottleneck for two-level methods like Parareal, [2] but not for a multilevel scheme like XBraid where the coarsest grid is of trivial size.
- By forming the coarse grid to have the same sparsity structure and time stepper as the fine grid, the algorithm can recur easily and efficiently.
- Interpolation is ideal or exact, in that an application of interpolation leaves a zero residual at all F-points.
- The process is applied recursively until a trivially sized temporal grid is reached, e.g., 2 or 3 time points. Thus, the coarsening rate m determines how many levels there are in the hierarchy. For instance in this figure, a 3 level hierarchy is shown. Three levels are chosen because there are six time points, $m = 2$ and $m^2 < 6 \leq m^3$. If the coarsening rate had been $m = 4$ then there would only be two levels because there would be no more points to coarsen!



By default, XBraid will subdivide the time domain into evenly sized time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes.

3.2 Overview of the XBraid Code

XBraid is designed to run in conjunction with an existing application code that can be wrapped per our interface. This application code will implement some time marching simulation like fluid flow. Essentially, the user has to take their application code and extract a stand-alone time-stepping function Φ that can evolve a solution from one time value to another, regardless of time step size. After this is done, the XBraid code takes care of the parallelism in the time dimension.

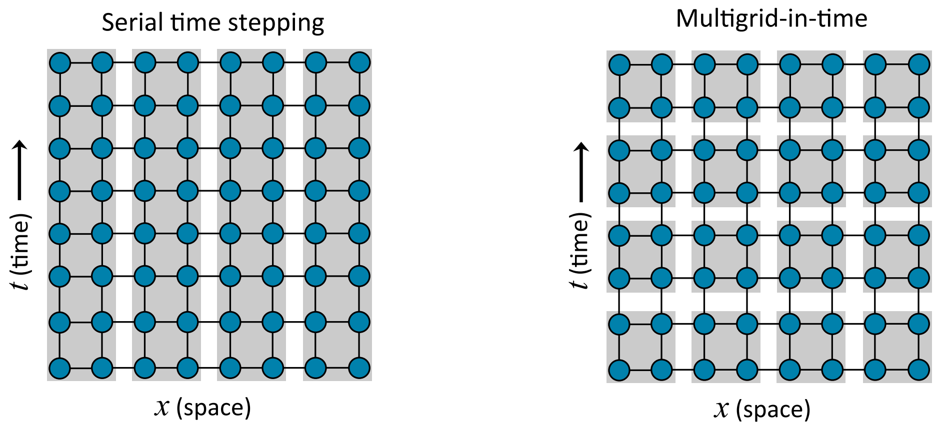
XBraid

- is written in C and can easily interface with Fortran, C++, and Python
- uses MPI for parallelism
- self documents through comments in the source code and through $*$.md files
- functions and structures are prefixed by *braid*
  - User routines are prefixed by `braid_`
  - Developer routines are prefixed by `_braid_`

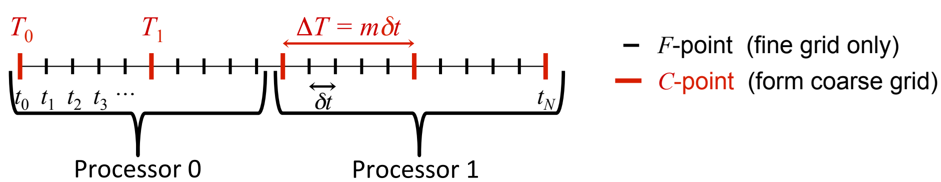
3.2.1 Parallel decomposition and memory

- XBraid decomposes the problem in parallel as depicted next. As you can see, traditional time stepping only stores one time step at a time, but only enjoys a spatial data decomposition and spatial parallelism. On the other hand, XBraid stores multiple time steps simultaneously and each processor holds a space-time chunk reflecting both the spatial and temporal parallelism.

[2] Lions, J., Yvon Maday, and Gabriel Turinici. "A"parareal"in time discretization of PDE's." Comptes Rendus de l'Academie des Sciences Series I Mathematics 332.7 (2001): 661-668.



- XBraid only handles temporal parallelism and is agnostic to the spatial decomposition. See braid_SplitCommworld. Each processor owns a certain number of CF intervals of points. In the following figure, processor 1 and processor 2 each own 2 CF intervals. XBraid distributes intervals evenly on the finest grid.



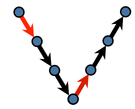
- XBraid increases the parallelism significantly, but now several time steps need to be stored, requiring more memory. XBraid employs two strategies to address the increased memory costs.
  - First, one need not solve the whole problem at once. Storing only one space-time slab is advisable. That is, solve for as many time steps (say *k* time steps) as you have available memory for. Then move on to the next *k* time steps.
  - Second, XBraid provides support for storing only C-points. Whenever an F-point is needed, it is generated by F-relaxation. More precisely, only the red C-point time values in the previous figure are stored. Coarsening is usually aggressive with $m = 8, 16, 32, ...$, so the storage requirements of XBraid are significantly reduced when compared to storing all of the time values.

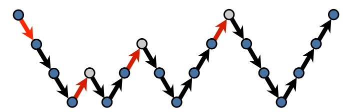
  Overall, the memory multiplier per processor when using XBraid is $O(1)$ if space-time coarsening (see The Simplest Example) is used and $O(\log_m N)$ for time-only coarsening. The time-only coarsening option is the default and requires no user-written spatial interpolation/restriction routines (which is the case for space-time coasrening). We note that the base of the logarithm is m, which can be quite large.

3.2.2 Cycling and relaxation strategies

There are two main cycling strategies available in XBraid, F-and V-cycles. These two cycles differ in how often and the order in which coarse levels are visited. A V-cycle is depicted next, and is a simple recursive application of the Two-Grid Algorithm.

An F-cycle visits coarse grids more frequently and in a different order. Essentially, an F-cycle uses a V-cycle as the post-smoother, which is an expensive choice for relaxation. But, this extra work gives you a closer approximation to a two-grid cycle, and a faster convergence rate at the extra expense of more work. The effectiveness of a V-cycle as a relaxation scheme can be seen in Figure 2, where one V-cycle globally propagates and *smoothes* the error. The cycling strategy of an F-cycle is depicted next.





Next, we make a few points about F- versus V-cycles.

- One V-cycle iteration is cheaper than one F-cycle iteration.
- But, F-cycles often converge more quickly. For some test cases, this difference can be quite large. The cycle choice for the best time to solution will be problem dependent. See Scaling Study with this Example for a case study of cycling strategies.
- For exceptionally strong F-cycles, the option braid_SetNFMGVcyc can be set to use multiple V-cycles as relaxation. This has proven useful for some problems with a strongly advective nature.

The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

- Using FCF, FCFCF, or FCFCFCF relaxation corresponds to passing *braid_SetNRelax* a value of 1, 2 or 3 respectively, and will result in an XBraid cycle that converges more quickly as the number of relaxations grows.
- But as the number of relaxations grows, each XBraid cycle becomes more expensive. The optimal relaxation strategy for the best time to solution will be problem dependent.
- However, a good first step is to try FCF on all levels (i.e., *braid_SetNRelax(core, -1, 1)*).
- A common optimization is to first set FCF on all levels (i.e., *braid_setnrelax(core, -1, 1)*), but then overwrite the FCF option on level 0 so that only F-relaxation is done on level 0, (i.e., *braid_setnrelax(core, 0, 1)*). Another strategy is to use F-relaxation on all levels together with F-cycles.
- See Scaling Study with this Example for a case study of relaxation strategies.

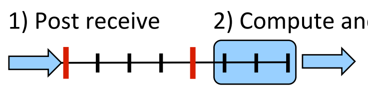
There is also a weighted relaxation option, which applies weighted-Jacobi at the C-points during the C-relaxation. Experiments with the 1D heat equation and 1D advection showed iteration gains of 10-25% for V-cycles when the experimentally optimal weight was used.

- For the heat equation, a weight of around 1.3 was experimentally optimal
- For the advection equation, weights between 1.4 and 1.8 were experimentally optimal
- Set this option with braid_SetCRelaxWt, which allows you to set a global relaxation weight, or an individual weight for each level. In general, under-relaxation (weight $<$ 1.0) never improved performance, but over-relxation (1.0 $<$ weight $<$ 2.0) often offered some improvement.

Last, `Parallel Time Integration with Multigrid` has a more in depth case study of cycling and relaxation strategies

3.2.3 Overlapping communication and computation

XBraid effectively overlaps communication and computation. The main computational kernel of XBraid is one relaxation sweep touching all the CF intervals. At the start of a relaxation sweep, each process first posts a non-blocking receive at its left-most point. It then carries out F-relaxation in each interval, starting with the right-most interval to send the data to the neighboring process as soon as possible. If each process has multiple CF intervals at this XBraid level, the strategy allows for complete overlap.



3.2.4 Configuring the XBraid Hierarchy

Some of the more basic XBraid function calls allow you to control aspects discussed here.

- braid_SetFMG: switches between using F- and V-cycles.
- braid_SetMaxIter: sets the maximum number of XBraid iterations
- braid_SetCFactor: sets the coarsening factor for any (or all levels)
- braid_SetNRelax: sets the number of CF-relaxation sweeps for any (or all levels)
- braid_SetRelTol, braid_SetAbsTol: sets the stopping tolerance
- braid_SetMinCoarse: sets the minimum possible coarse grid size
- braid_SetMaxLevels: sets the maximum number of levels in the XBraid hierarchy

3.2.5 Halting tolerance

Another important configuration aspect regards setting a residual halting tolerance. Setting a tolerance involves these three XBraid options:

1. braid_PtFcnSpatialNorm

   This user-defined function carries out a spatial norm by taking the norm of a braid_Vector. A common choice is the standard Eucliden norm (2-norm), but many other choices are possible, such as an L2-norm based on a finite element space.

2. braid_SetTemporalNorm

   This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by braid_PtFcnSpatialNorm at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.

   There are three *tnorm* options supported by braid_SetTemporalNorm. We let the summation index *i* be over all C-point values on the fine time grid, *k* refer to the current XBraid iteration, *r* be residual values, *space_time* norms be a norm over the entire space-time domain and *spatial_norm* be the user-defined spatial norm from braid_PtFcnSpatialNorm. Thus, r_i is the residual at the *ith* C-point, and $r^{(k)}$ is the residual at the *kth* XBraid iteration. The three options are then defined as,

- *tnorm=1*: One-norm summation of spatial norms

$$\|r^{(k)}\|_{\text{space\_time}} = \Sigma_i \|r_i^{(k)}\|_{\text{spatial\_norm}}$$

  If braid_PtFcnSpatialNorm is the one-norm over space, then this is equivalent to the one-norm of the global space-time residual vector.

- *tnorm=2*: Two-norm summation of spatial norms

$$\|r^{(k)}\|_{\text{space\_time}} = \left(\Sigma_i \|r_i^{(k)}\|^2_{\text{spatial\_norm}}\right)^{1/2}$$

  If braid_PtFcnSpatialNorm is the Euclidean norm (two-norm) over space, then this is equivalent to the Euclidean-norm of the global space-time residual vector.

- *tnorm=3*: Infinity-norm combination of spatial norms

$$\|r^{(k)}\|_{\text{space\_time}} = \max_i \|r_i^{(k)}\|_{\text{spatial\_norm}}$$

  If braid_PtFcnSpatialNorm is the infinity-norm over space, then this is equivalent to the infinity-norm of the global space-time residual vector.

The default choice is *tnorm=2*

3. braid_SetAbsTol, braid_SetRelTol

   - If an absolute tolerance is used, then

$$\|r^{(k)}\|_{\text{space\_time}} < \text{tol}$$

     defines when to halt.

   - If a relative tolerance is used, then

$$\frac{\|r^{(k)}\|_{\text{space\_time}}}{\|r^{(0)}\|_{\text{space\_time}}} < \text{tol}$$

     defines when to halt. That is, the current *kth* residual is scaled by the initial residual before comparison to the halting tolerance. This is similar to typical relative residual halting tolerances used in spatial multigrid, but can be a dangerous choice in this setting.

Care should be practiced when choosing a halting tolerance. For instance, if a relative tolerance is used, then issues can arise when the initial guess is zero for large numbers of time steps. Taking the case where the initial guess (defined by braid_PtFcnInit) is 0 for all time values $t > 0$, the initial residual norm will essentially only be nonzero at the first time value,

$$\|r^{(0)}\|_{\text{space\_time}} \approx \|r_1^{(k)}\|_{\text{spatial\_norm}}$$

This will skew the relative halting tolerance, especially if the number of time steps increases, but the initial residual norm does not.

A better strategy is to choose an absolute tolerance that takes your space-time domain size into account, as in Section Scaling Study with this Example, or to use an infinity-norm temporal norm option.

3.2.6 Debugging XBraid

Wrapping and debugging a code with XBraid typically follows a few steps.

- Test your wrapped functions with XBraid test functions, e.g., braid_TestClone or braid_TestSum.

- Set max levels to 1 (braid_SetMaxLevels) and run an XBraid simulation. You should get the exact same answer as that achieved with sequential time stepping. If you make sure that the time-grids used by XBraid and by sequential time stepping are bit-wise the same (by using the user-defined time grid option braid_SetTimeGrid), then the agreement of their solutions should be bit-wise the same.
- Continue with max levels equal to 1, but switch to two processors in time. Check that the answer again exactly matches sequential time stepping. This test checks that the information in braid_Vector is sufficient to correctly start the simulation on the second processor in time.
- Set max levels to 2, halting tolerance to 0.0 (braid_SetAbsTol), max iterations to 3 (braid_SetMaxIter) and turn on the option braid_SetSeqSoln. This will use the solution from sequential time-stepping as the initial guess for XBraid and then run 3 iterations. The residual should be exactly 0 each iteration, verifying the fixed-point nature of XBraid and a (hopefully!) correct implementation. The residual may be on the order of machine epsilon (or smaller). Repeat this test for multiple processors in time (and space if possible).
- A similar test turns on debug level printing by passing a print level of 3 to braid_SetPrintLevel. This will print out the residual norm at each C-point. XBraid with FCF-relaxation has the property that the exact solution is propagated forward two C-points each iteration. Thus, this should be reflected by numerically zero residual values for the first so many time points. Repeat this test for multiple processors in time (and space if possible).
- Finally, run some multilevel tests, making sure that the XBraid results are within the halting tolerance of the solutions generated by sequential time-stepping. Repeat this test for multiple processors in time (and space if possible).
- Congratulations! Your code is now verified.

3.3 Computing Derivatives with XBraid_Adjoint

XBraid_Adjoint has been developed in collaboration with the Scientific Computing group at TU Kaiserslautern, Germany, and in particular with Dr. Stefanie Guenther and Prof. Nicolas Gauger.

In many application scenarios, the ODE system is driven by some independent design parameters ρ. These can be any time-dependent or time-independent parameters that uniquely determine the solution of the ODE (e.g. a boundary condition, material coefficients, etc.). In a discretized ODE setting, the user's time-stepping routine might then be written as

$$u_i = \Phi_i(u_{i-1}, \rho), \quad \forall i = 1, \dots N,$$

where the time-stepper Φ_i, which propagates a state u_{i-1} at a time t_{i-1} to the next time step at t_i, now also depends on the design parameters ρ. In order to quantify the simulation output for the given design, a real-valued objective function can then be set up that measures the quality of the ODE solution:

$$J(\mathbf{u}, \rho) \in \mathrm{R}.$$

Here, $\mathbf{u} = (u_0, \dots, u_N)$ denotes the space-time state solution for a given design.

XBraid_Adjoint is a consistent discrete time-parallel adjoint solver for XBraid which provides sensitivity information of the output quantity J with respect to the user-defined design parameters ρ. The ability to compute sensitivities can greatly improve and enhance the simulation tool, for example for solving

- Design optimization problems,
- Optimal control problems,
- Parameter estimation for validation and verification purposes,
- Error estimation,
- Uncertainty quantification techniques.

XBraid_Adjoint is non-intrusive with respect to the adjoint time-stepping scheme so that existing time-serial adjoint codes can be integrated easily though an extended user-interface.

3.3.1 Short Introduction to Adjoint-based Sensitivity Computation

Adjoint-based sensitivities compute the total derivative of J with respect to changes in the design parameters ρ by solving additional so-called adjoint equations. We will briefly introduce the idea in the following. You can skip this section, if you are familiar with adjoint sensitivity computation in general and move to Overview of the XBraid_Adjoint Algorithm immedately. Information on the adjoint method can be found in [Giles, Pierce, 2000] [3] amongst many others.

Consider an augmented (so-called *Lagrange*) funtion

$$L(\mathbf{u},\rho) = J(\mathbf{u},\rho) + \bar{\mathbf{u}}^T A(\mathbf{u},\rho)$$

where the discretized time-stepping ODE equations in

$$A(\mathbf{u},\rho) := \begin{pmatrix} \Phi_1(u_0,\rho) - u_1 \\ \vdots \\ \Phi_N(u_{N-1},\rho) - u_N \end{pmatrix}$$

have been added to the objective function, and multiplied with so-called *adjoint* variables $\bar{\mathbf{u}} = (\bar{u}_1, \dots, \bar{u}_N)$. Since the added term is zero for all design and state variables that satisfy the discrete ODE equations, the total derivative of J and L with respect to the design match. Using the chain rule of differentiation, this derivative can be expressed as

$$\frac{\mathrm{d}J}{\mathrm{d}\rho} = \frac{\mathrm{d}L}{\mathrm{d}\rho} = \frac{\partial J}{\partial \mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\rho} + \frac{\partial J}{\partial \rho} + \bar{\mathbf{u}}^T\left(\frac{\partial A}{\partial \mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\rho} + \frac{\partial A}{\partial \rho}\right)$$

where ∂ denotes partial derivatives – in contrast to the total derivative (i.e. the sensitivity) denoted by d.

When computing this derivative, the terms in red are the ones that are computationally most expensive. In fact, the cost for computing these sensitivities scale linearly with the number of design parameters, i.e. the dimension of ρ. These costs can grow quickly. For example, consider a finite differencing setting, where a re-computation of the entire space-time state would be necessary for each design variable, because a perturbation of the design must be computed in all the unit directions of the design space. In order to avoid these costs, the adjoint method aims to set the adjoint variable $\bar{\mathbf{u}}$ such that these red terms add up to zero in the above expression. Hence, if we solve first for

$$\left(\frac{\partial J}{\partial \mathbf{u}}\right)^T + \left(\frac{\partial A}{\partial \mathbf{u}}\right)^T \bar{\mathbf{u}} = 0$$

for the adjoint variable $\bar{\mathbf{u}}$, then the so-called *reduced gradient* of J, which is the transpose of the total derivative of J with respect to the design, is given by

$$\left(\frac{\mathrm{d}J}{\mathrm{d}\rho}\right)^T = \left(\frac{\partial J}{\partial \rho}\right)^T + \left(\frac{\partial A}{\partial \rho}\right)^T \bar{\mathbf{u}}$$

The advantage of this strategy is, that in order to compute the sensitivity of J with respect to ρ, only one additional space-time equation (adjoint) for $\bar{\mathbf{u}}$ has to be solved, in addition to evaluating the partial derivatives. The computational cost for computing $\mathrm{d}J/\mathrm{d}\rho$ therefore does not scale in this setting with the number of design parameters.

For the time-dependent discrete ODE problem, the adjoint equation from above reads

unsteady adjoint: $$\bar{u}_i = \partial_{u_i} J(\mathbf{u},\rho)^T + \left(\partial_{u_i}\Phi_{i+1}(u_i,\rho)\right)^T \bar{u}_{i+1} \qquad \forall i = N \dots, 1$$

using the terminal condition $u_{N+1} := 0$. The reduced gradient is given by

reduced gradient: $$\left(\frac{\partial J}{\partial \rho}\right)^T = \partial_\rho J(\mathbf{u},\rho)^T + \sum_{i=1}^{N} \left(\partial_\rho \Phi_i(u_{i-1},\rho)\right)^T \bar{u}_i$$

[3] Giles, M.B., Pierce, N.A.: "An introduction to the adjoint approach to design." Flow, Turbulence and Combustion 65(3), 393–415 (2000)

3.3.2 Overview of the XBraid_Adjoint Algorithm

The unsteady adjoint equations can in principle be solved “backwards in time" in a time-serial manner, starting from the terminal condition $\bar{u}_{N+1} = 0$. However, the parallel-in-time XBraid_Adjoint solver offers speedup by distributing the backwards-in-time phase onto multiple processors along the time domain. Its implementation is based on techniques of the reverse-mode of Automatic Differentiation applied to one primal XBraid iteration. To that end, each primal iteration is augmented by an objective function evaluation, followed by updates for the space-time adjoint variable $\bar{\mathbf{u}}$, as well as evaluation of the reduced gradient denoted by $\bar{\rho}$. In particular, the following so-called *piggy-back* iteration is performed:

1. **XBraid**: update the state and evaluate the objective function

$$\mathbf{u}^{(k+1)} \leftarrow \text{XBraid}(\mathbf{u}^{(k)}, \rho), \quad J \leftarrow J(\mathbf{u}^{(k)}, \rho)$$

2. **XBraid_Adjoint**: update the adjoint and evaluate the reduced gradient

$$\bar{\mathbf{u}}^{(k+1)} \leftarrow \text{XBraid\_Adjoint}(\mathbf{u}^{(k)}, \bar{\mathbf{u}}^{(k)}, \rho), \quad \bar{\rho} \leftarrow \left(\frac{\mathrm{d}J(\mathbf{u}^{(k)}, \rho)}{\mathrm{d}\rho} \right)^T$$

Each XBraid_Adjoint iteration moves backwards though the primal XBraid multigrid cycle. It collects local partial derivatives of the elemental XBraid operations in reverse order and concatenates them using the chain rule of differentiation. This is the basic idea of the reverse mode of Automatic Differentiation (AD). This yields a consistent discrete time-parallel adjoint solver that inherits the parallel scaling properties of the primal XBraid solver.

Further, XBraid_Adjoint is non-intrusive for existing adjoint methods based on sequential time marching schemes. It adds additional user-defined routines to the primal XBraid interface, in order to define the propagation of sensitivities of the forward time stepper backwards-in-time and the evaluation of partial derivatives of the local objective function at each time step. In cases where a time-serial unsteady adjoint solver is already available, this backwards time stepping capability can be easily wrapped according to the adjoint user interface with little extra coding.

The adjoint solve in the above piggy-back iteration converges at the same convergence rate as the primal state variables. However since the adjoint equations depend on the state solution, the adjoint convergence will slightly lag behind the convergence of the state. More information on convergence results and implementational details for XBraid_Adjoint can be found in [Gunther, Gauger, Schroder, 2017]. [4]

3.3.3 Overview of the XBraid_Adjoint Code

XBraid_Adjoint offers a non-intrusive approach for time-parallelization of existing time-serial adjoint codes. To that end, an extended user-interface allows the user to wrap their existing code for evaluating the objective function and performing a backwards-in-time adjoint step into routines according to the XBraid_Adjoint interface.

3.3.3.1 Objective function evaluation

The user-interface for XBraid_Adjoint allows for objective functions of the following type:

$$J = F\left(\int_{t_0}^{t^1} f(u(t), \rho) \, \mathrm{d}t \right).$$

This involves a time-integral part of some time-dependent quantity of interest f as well as a *postprocessing* function F. The time-interval boundaries t_0, t_1 can be set using the options braid_SetTStartObjective and braid_SetTStopObjective, otherwise the entire time domain will be considered. Note that these options can be used for objective functions that are only evaluated at one specific time instance by setting $t_0 = t_1$ (e.g. in cases where only the last time step is of interest). The postprocessing function F offers the possibility to further modify the time-integral, e.g. for setting up a

[4] Günther, S., Gauger, N.R. and Schroder, J.B. ”A Non-Intrusive Parallel-in-Time Adjoint Solver with the XBraid Library.” Computing and Visualization in Science, Springer, (accepted), (2017)

tracking-type objective function (substract a target value and square), or for adding relaxation or penalty terms. While defining f is mandatory for XBraid_Adjoint, the postprocessing routine F is optional and is passed to XBraid_Adjoint though the optional braid_SetPostprocessObjective and braid_SetPostprocessObjective_diff routines. XBraid_Adjoint will perform the time-integration by summing up the f evaluations in the given time-domain

$$I \leftarrow \sum_{i=i_0}^{i_1} f(u_i, \rho)$$

followed by a call to the postprocessing function F, if set:

$$J \leftarrow F(I, \rho)\,.$$

Note that any integration rule for computing I, e.g. for scaling contributions from $f()$, must be done by the user.

Partial derivatives of user-routines

The user needs to provide the derivatives of the time-stepper Φ and function evaluation f (and potentially F) for X↩ Braid_Adjoint. Those are provided in terms of transposed matrix-vector products in the following way:

1. **Derivatives of the objective function** J:
   - **Time-dependent part** f: The user provides a routine that evaluates the following transposed partial derivatives of f multiplied with the scalar input $\bar{F}$:

     $$\bar{u}_i \leftarrow \left(\frac{\partial f(u_i,\rho)}{\partial u_i}\right)^T \bar{F}$$

     $$\bar{\rho} \leftarrow \bar{\rho} + \left(\frac{\partial f(u_i,\rho)}{\partial \rho}\right)^T \bar{F}$$

     The scalar input $\bar{F}$ equals 1.0, if no postpocessing function F has been set.
   - **Postprocessing** F: If the postprocessing routine has been set, the user needs to provide it's transposed partial derivatives in the following way:

     $$\bar{F} \leftarrow \frac{\partial F(I,\rho)}{\partial I}$$

     $$\bar{\rho} \leftarrow \rho + \frac{\partial F(I,\rho)}{\partial \rho}$$

2. **Derivatives of the time-stepper** Φ_i: The user provides a routine that computes the following transposed partial derivatives of Φ_i multiplied with the adjoint input vector $\bar{u}_i$:

   $$\bar{u}_i \leftarrow \left(\frac{\partial \Phi(u_i,\rho)}{\partial u_i}\right)^T \bar{u}_i$$

   $$\bar{\rho} \leftarrow \bar{\rho} + \left(\frac{\partial \Phi(u_i,\rho)}{\partial \rho}\right)^T \bar{u}_i$$

Note that the partial derivatives with respect to ρ always *update* the reduced gradient $\bar{\rho}$ instead of overwriting it (i.e. they are a plus-equal operation, $+=$). Therefore, the gradient needs to be reset to zero before each iteration of XBraid_↩ Adjoint, which is taken care of by XBraid_Adjoint calling an additional user-defined routine braid_PtFcnResetGradient.

Depending on the nature of the design variables, it is neccessary to gather gradient information in $\bar{\rho}$ from all time-processors after XBraid_Adjoint has finished. It is the user's responsibility to do that, if needed, e.g. through a call to MPI_Allreduce.

Halting tolerance

Similar to the primal XBraid algorithm, the user can choose a halting tolerance for XBraid_Adjoint which is based on the adjoint residual norm. An absolute tolerance (braid_SetAbsTolAdjoint)

$$\|\bar{\mathbf{u}}^{(k)} - \bar{\mathbf{u}}^{(k-1)}\|_{\text{space\_time}} < \text{tol\_adjoint}$$

or a relative tolerance (braid_SetRelTolAdjoint)

$$\frac{\|\bar{\mathbf{u}}^{(k)} - \bar{\mathbf{u}}^{(k-1)}\|_{\text{space\_time}}}{\|\bar{\mathbf{u}}^{(1)} - \bar{\mathbf{u}}^{(0)}\|_{\text{space\_time}}} < \text{tol\_adjoint}$$

can be chosen.

Finite Difference Testing

You can verify the gradient computed from XBraid_Adjoint using Finite Differences. Let e_i denote the i-th unit vector in the design space, then the i-th entry of the gradient should match with

$$i\text{-th Finite Difference:} \quad \frac{J(\mathbf{u}_{\rho+he_i}, \rho + he_i) - J(\mathbf{u}, \rho)}{h}$$

for a small perturbation $h > 0$. Here, $\mathbf{u}_{\rho+he_i}$ denotes the new state solution for the perturbed design variable. Keep in mind, that round-off errors have to be considered when computing the Finite Differences for very small perturbations $h \to 0$. Hence, you should vary the parameter to find the best fit.

In order to save some computational work while computing the perturbed objective function value, XBraid_Adjoint can run in `ObjectiveOnly` mode, see braid_SetObjectiveOnly. When in this mode, XBraid_Adjoint will only solve the ODE system and evaluate the objective function, without actually computing its derivative. This option might also be useful within an optimization framework e.g. for implementing a line-search procedure.

Getting started

- Look at the simple example Simplest XBraid_Adjoint example in order to get started. This example is in `examples/ex-01-adjoint.c`, which implements XBraid_Adjoint sensitivity computation for a scalar ODE.

3.4 Citing XBraid

To cite XBraid, please state in your text the version number from the VERSION file, and please cite the project website in your bibliography as

> [1] XBraid: Parallel multigrid in time. `http://llnl.gov/casc/xbraid`.

The corresponding BibTex entry is

```
@misc{xbraid-package,
  title = {{XB}raid: Parallel multigrid in time},
  howpublished = {\url{http://llnl.gov/casc/xbraid}}
  }
```

3.5 Summary

- XBraid applies multigrid to the time dimension.
  - This exposes concurrency in the time dimension.
  - The potential for speedup is large, 10x, 100x, ...

- This is a non-intrusive approach, with an unchanged time discretization defined by user.
- Parallel time integration is only useful beyond some scale. This is evidenced by the experimental results below. For smaller numbers of cores sequential time stepping is faster, but at larger core counts XBraid is much faster.
- The more time steps that you can parallelize over, the better your speedup will be.
- XBraid is optimal for a variety of parabolic problems (see the examples directory).
- XBraid_Adjoint provides time-parallel adjoint-based sensitivities of output quantities with respect to user-defined design variables
  - It is non-intrusive with respect to existing adjoint time-marching schemes
  - It inherits parallel scaling properties from XBraid

4 Examples

This section is the chief *tutorial* of XBraid, illustrating how to use it through a sequence of progressively more sophisticated examples.

4.1 The Simplest Example

User Defined Structures and Wrappers

The user must wrap their existing time stepping routine per the XBraid interface. To do this, the user must define two data structures and some wrapper routines. To make the idea more concrete, we now give these function definitions from `examples/ex-01`, which implements a scalar ODE,

$$u_t = \lambda u.$$

The two data structures are:

1. **App**: This holds a wide variety of information and is *global* in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. Here for illustration, this is just an integer storing a processor's rank.

```
typedef struct _braid_App_struct
{
   int       rank;
} my_App;
```

2. **Vector**: this defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information. Here, the vector is just a scalar double.

```
typedef struct _braid_Vector_struct
{
   double value;
} my_Vector;
```

The user must also define a few wrapper routines. Note, that the *app* structure is the first argument to every function.

1. **Step**: This function tells XBraid how to take a time step, and is the core user routine. The user must advance the vector *u* from time *tstart* to time *tstop*. Note how the time values are given to the user through the *status* structure

and associated *Get* routine. **Important note:** the g_i function from Overview of the XBraid Algorithm must be incorporated into *Step*, so that the following equation is solved by default.

$$\Phi(u_i) = 0.$$

The *ustop* parameter serves as an approximation to the solution at time *tstop* and is not needed here. It can be useful for implicit schemes that require an initial guess for a linear or nonlinear solver. The use of *fstop* is an advanced parameter (not required) and forms the the right-hand side of the nonlinear problem on the given time grid. This value is only nonzero when providing a residual with braid_SetResidual. More information on how to use this optional feature is given below.

Here advancing the solution just involves the scalar λ.

```
int
my_Step(braid_App        app,
        braid_Vector     ustop,
        braid_Vector     fstop,
        braid_Vector     u,
        braid_StepStatus status)
{
   double tstart;             /* current time */
   double tstop;              /* evolve to this time*/
   braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

   /* Use backward Euler to propagate solution */
   (u->value) = 1./(1. + tstop-tstart)*(u->value);

   return 0;
}
```

2. **Init**: This function tells XBraid how to initialize a vector at time *t*. Here that is just allocating and setting a scalar on the heap.

```
int
my_Init(braid_App     app,
        double        t,
        braid_Vector *u_ptr)
{
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   if (t == 0.0) /* Initial condition */
   {
      (u->value) = 1.0;
   }
   else /* All other time points set to arbitrary value */
   {
      (u->value) = 0.456;
   }
   *u_ptr = u;

   return 0;
}
```

3. **Clone**: This function tells XBraid how to clone a vector into a new vector.

```
int
my_Clone(braid_App     app,
         braid_Vector  u,
         braid_Vector *v_ptr)
{
   my_Vector *v;

   v = (my_Vector *) malloc(sizeof(my_Vector));
   (v->value) = (u->value);
```

```
   *v_ptr = v;

   return 0;
}
```

4. **Free**: This function tells XBraid how to free a vector.

```
int
my_Free(braid_App    app,
        braid_Vector u)
{
   free(u);

   return 0;
}
```

5. **Sum**: This function tells XBraid how to sum two vectors (AXPY operation).

```
int
my_Sum(braid_App    app,
       double       alpha,
       braid_Vector x,
       double       beta,
       braid_Vector y)
{
   (y->value) = alpha*(x->value) + beta*(y->value);

   return 0;
}
```

6. **SpatialNorm**: This function tells XBraid how to take the norm of a *braid_Vector* and is used for halting. This norm is only over space. A common norm choice is the standard Euclidean norm, but many other choices are possible, such as an L2-norm based on a finite element space. The norm choice should be based on what makes sense for your problem. How to accumulate spatial norm values to obtain a global space-time residual norm for halting decisions is controlled by braid_SetTemporalNorm.

```
int
my_SpatialNorm(braid_App     app,
               braid_Vector  u,
               double       *norm_ptr)
{
   double dot;

   dot = (u->value)*(u->value);
   *norm_ptr = sqrt(dot);

   return 0;
}
```

7. **Access**: This function allows the user access to XBraid and the current solution vector at time *t*. This is most commonly used to print solution(s) to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value *t* of the vector *u* and even more information in *astatus*. This lets you tailor the output to only certain time values at certain XBraid iterations. Querying *astatus* for such information is done through *braid_AccessStatusGet***(..)* routines.

   The frequency of the calls to *access* is controlled through braid_SetAccessLevel. For instance, if *access_level* is set to 2, then *access* is called every XBraid iteration and on every XBraid level. In this case, querying *astatus* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation. The default *access_level* is 1 and gives the user access only after the simulation ends and only on the finest time-grid.

   Eventually, this routine will allow for broader access to XBraid and computational steering.

See `examples/ex-03` and `drivers/drive-diffusion` for more advanced uses of the *access* function. In `drive-diffusion`, *access* is used to write solution vectors to a GLVIS visualization port, and `ex-03` uses *access* to write to .vtu files.

```
int
my_Access(braid_App          app,
          braid_Vector       u,
          braid_AccessStatus astatus)
{
   int        index;
   char       filename[255];
   FILE      *file;

   braid_AccessStatusGetTIndex(astatus, &index);
   sprintf(filename, "%s.%04d.%03d", "ex-01.out", index, app->rank);
   file = fopen(filename, "w");
   fprintf(file, "%.14e\n", (u->value));
   fflush(file);
   fclose(file);

   return 0;
}
```

8. **BufSize**, **BufPack**, **BufUnpack**: These three routines tell XBraid how to communicate vectors between processors. *BufPack* packs a vector into a `void *` buffer for MPI and then *BufUnPack* unpacks the `void *` buffer into a vector. Here doing that for a scalar is trivial. *BufSize* computes the upper bound for the size of an arbitrary vector.

   Note how *BufPack* also sets the size in *bstatus*. This value is optional, but if set it should be the exact number of bytes packed, while *BufSize* should provide only an upper-bound on a possible buffer size. This flexibility allows for the buffer to be allocated the fewest possible times, but smaller messages to be sent when needed. For instance, this occurs when using variable spatial grid sizes. **To avoid MPI issues, it is very important that BufSize be pessimistic, provide an upper bound, and return the same value across processors.**

   In general, the buffer should be self-contained. The receiving processor should be able to pull all necessary information from the buffer in order to properly interpret and unpack the buffer.

```
int
my_BufSize(braid_App          app,
           int                *size_ptr,
           braid_BufferStatus bstatus)
{
   *size_ptr = sizeof(double);
   return 0;
}

int
my_BufPack(braid_App          app,
           braid_Vector       u,
           void               *buffer,
           braid_BufferStatus bstatus)
{
   double *dbuffer = buffer;

   dbuffer[0] = (u->value);
   braid_BufferStatusSetSize( bstatus, sizeof(double) );

   return 0;
}

int
my_BufUnpack(braid_App          app,
             void               *buffer,
             braid_Vector       *u_ptr,
             braid_BufferStatus bstatus)
```

```
{
   double    *dbuffer = buffer;
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   (u->value) = dbuffer[0];
   *u_ptr = u;

   return 0;
}
```

4.1.1 Running XBraid for the Simplest Example

A typical flow of events in the *main* function is to first initialize the *app* structure.

```
/* set up app structure */
app = (my_App *) malloc(sizeof(my_App));
(app->rank)   = rank;
```

Then, the data structure definitions and wrapper routines are passed to XBraid. The core structure is used by XBraid for internal data structures.

```
braid_Core  core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
           my_Step, my_Init, my_Clone, my_Free, my_Sum, my_SpatialNorm,
           my_Access, my_BufSize, my_BufPack, my_BufUnpack, &core);
```

Then, XBraid options are set.

```
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetAbsTol(core, tol);
braid_SetCFactor(core, -1, cfactor);
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run ex-01, type

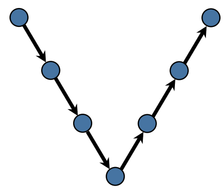
```
ex-01
```

4.2 Some Advanced Features

We now give an overview of some *optional* advanced features that will be implemented in some of the following examples.

1. **SCoarsen**, **SRestrict**: These are advanced options that allow for coarsening in space while you coarsen in time. This is useful for maintaining stable explicit schemes on coarse time scales and is not needed here. See `examples/ex-02` for a simple example of this feature, and then `drivers/drive-diffusion` and `drivers/drive-diffusion-2D` for more advanced examples of this feature.

   These functions allow you to vary the spatial mesh size on XBraid levels as depicted here where the spatial and temporal grid sizes are halved every level.



2. **Residual**: A user-defined residual can be provided with the function braid_SetResidual and can result in substantial computational savings, as explained below. However to use this advanced feature, one must first understand how XBraid measures the residual. XBraid computes residuals of this equation,

$$A_i(u_i, u_{i-1}) = f_i,$$

where $A_i(,)$ evaluates one block-row of the the global space-time operator A. The forcing f_i is the XBraid forcing, which is the FAS right-hand-side term on coarse grids and 0 on the finest grid. The PDE forcing goes inside of A_i.

Since XBraid assumes one-step methods, $A_i()$ is defined to be

$$A_i(u_i, u_{i-1}) = -\Phi(u_{i-1}) + \Psi(u_i),$$

i.e., the subdiagonal and diagonal blocks of A.

Default setting: In the default XBraid setting (no residual option used), the user only implements *Step()* and *Step()* will simply apply $\Phi()$, because $\Psi()$ is assumed to be the identity. Thus, XBraid can compute the residual using only the user-defined *Step()* function by combining *Step()* with the *Sum()* function, i.e.

$$r_i = f_i + \Phi(u_{i-1}) - u_i.$$

The *fstop* parameter in *Step()* corresponds to f_i, but is always passed in as NULL to the user in this setting and should be ignored. This is because XBraid can compute the contribution of f_i to the residual on its own using the *Sum()* function.

An implication of this is that the evaluation of $\Phi()$ on the finest grid must be very accurate, or the residual will not be accurate. This leads to a nonintrusive, but expensive algorithm. The accuracy of $\Phi()$ can be relaxed on coarser grids to save computations.

Residual setting: The alternative to the above default least-intrusive strategy is to have the user define

$$A_i(u_i, u_{i-1}) = -\Phi(u_{i-1}) + \Psi(u_i),$$

directly, which is what the *Residual* function implements (set with braid_PtFcnResidual). In other words, the user now defines each block-row of the space-time operator, rather than only defining $\Phi()$. The user *Residual()* function computes $A_i(u_i, u_{i-1})$ and XBraid then subtracts this from f_i to compute r_i.

However, more care must now be taken when defining the *Step()* function. In particular, the *fstop* value (i.e., the f_i value) must be taken into account. Essentially, the definition of *Step()* changes so that it no longer defines $\Phi()$, but instead defines a (possibly inexact) solve of the equation defined by

$$A_i(u_i, u_{i-1}) = f_i.$$

Thus, *Step()* must be compatible with *Residual()*. Expanding the previous equation, we say that *Step()* must now compute

$$u_i = \Psi^{-1}(f_i + \Phi(u_{i-1})).$$

It is clear that the *fstop* value (i.e., the f_i value) must now be given to the *Step()* function so that this equation can be solved by the user. In other words, *fstop* is now no longer NULL.

Essentially, one can think of *Residual()* as defining the equation, and *Step()* defining a preconditioner for that row of the equation, or an inexact solve for u_i.

As an example, let $\Psi = (I + \Delta t L)$, where L is a Laplacian and $\Phi = I$. The application of the residual function will only be a sparse matrix-vector multiply, as opposed to the default case where an inversion is required for $\Phi = (I + \Delta t L)^{-1}$ and $\Psi = I$. This results in considerable computational savings. Moreover, the application of *Step()* now involves an inexact inversion of Ψ, e.g., by using just one spatial multigrid V-cycle. This again results in substantial computation savings when compared with the naive approach of a full matrix inversion.

Another way to think about the compatibility between Ψ and Φ is that

$$f_i - A_i(u_i, u_{i-1}) = 0$$

must hold exactly if u_i is an exact propagation of u_{i-1}, that is,

$$f_i - A_i(Step(u_{i-1}, f_i), u_{i-1}) = 0$$

must hold. When the accuracy of the *Step()* function is reduced (as mentioned above), this exact equality with 0 is lost, but this should evaluate to something `small`. There is an XBraid test function braid_TestResidual that tests for this compatibility.

The residual feature is implemented in the examples `examples/ex-01-expanded.c`, `examples/ex-02.c`, and `examples/ex-03.c`.

3. **Adaptive and variable time stepping**: This feature is available by first calling the function braid_SetRefine in the main driver and then using braid_StepStatusSetRFactor in the *Step* routine to set a refinement factor for interval [*tstart*, *tstop*]. In this way, user-defined criteria can subdivide intervals on the fly and adaptively refine in time. For instance, returning a refinement factor of 4 in *Step* will tell XBraid to subdivide that interval into 4 evenly spaced smaller intervals for the next iteration. Refinement can only be done on the finest XBraid level.

   The final time grid is constructed adaptively in an FMG-like cycle by refining the initial grid according to the requested refinement factors. Refinement stops when the requested factors are all one or when various upper bounds are reached such as the max number of time points or max number of time grid refinement levels allowed. No restriction on the refinement factors is applied within XBraid, so the user may want to apply his own upper bound on the refinement factors to avoid over-refinement. See `examples/ex-01-refinement.c` and `examples/ex-03.c` for an implementation of this.

4. **Richardson-based Error Estimation and Extrapolation**: This feature allows the user to access built-in Richardson-based error estimates and accuracy improving extrapolation. The error estimates and/or extrapolation can be turned on by using braid_SetRichardsonEstimation . Moreover, this feature can be used in conjunction with the above discussed function, braid_StepStatusSetRFactor, to achieve easy-to-use adaptive refinement in time.

   Essentially, Richardson extrapolation (RE) is used to improve the accuracy of the solution at the C-points on the finest level. When the built-in error estimate option is turned on, RE is used to estimate the local truncation error at each point. These estimates can be accessed through StepStatus and AccessStatus functions.

   The Richardson-based error estimates and extrapolation are only available after the first Braid iteration, in that the coarse level solution must be available to compute the error estimate and/or extrapolation. Thus, after an adaptive refinement (and new hierarchy is constructed), another iteration is again required for the error estimates to be available. If the error estimate isn't available, Braid returns a value of -1. See this example for more details

   ```
   examples/ex-06.c
   ```

5. **Shell-vector**: This feature supports the use of multi-step methods. The strategy for BDF-K methods is to allow for the lumping of `k` time points into a single XBraid vector. So, if the problem had 100 time points and the time-stepper was BDF-2, then XBraid would only `see` 50 time points but each XBraid vector would contain two separate time points. By lumping 2 time points into one vector, the BDF-2 scheme remains one-step and compatible with XBraid.

   However, the time-point spacing between the two points internal to the vector stays the same on all time grids, while the spacing between vectors grows on coarse time grids. This creates an irregular spacing which is problematic for BDF-k methods. Thus the shell-vector strategy lets meta-data be stored at all time points, even for F-points which are usually not stored, so that the irregular spacings can be tracked and accounted for with the BDF method. (Note, there are other possible uses for shell-vectors.)

   There are many strategies for handling the coarse time-grids with BDF methods (dropping the BDF order, adjusting time-point spacings inside the lumped vectors, etc...). Prospective users are encouraged to contact the devlopers through the XBraid Github page and issue tracker. This area is active research.

   See `examples/ex-01-expanded-bdf2.c`.

6. **Storage**: This option (see braid_SetStorage) allows the user to specify storage at all time points (C and F) or only at C-points. This extra storage is useful for implicit methods, where the solution value from the *previous XBraid iteration* for time step i can be used as the initial guess when computing step i with the implicit solver. This is often a better initial guess than using the solution value from the previous time step $i-1$. The default is to store only C-point values, thus the better initial guess is only available at C-points in the default setting. When storage is turned on at F-points, the better initial guess becomes available everywhere.

   In general, the user should always use the *ustop* parameter in *Step()* as the initial guess for an implicit solve. If storage is turned on (i.e., set to 0), then this value will always be the improved initial guess for C- and F-points. If storage is not turned on, then this will be the improved guess only for C-points. For F-points, it will equal the solution from the previous time step.

   See `examples/ex-03` for an example which uses this feature.

4.3 Simplest example expanded

These examples build on The Simplest Example, but still solve the scalar ODE,

$$u_t = \lambda u.$$

The goal here is to show more advanced features of XBraid.

- `examples/ex-01-expanded.c`: same as `ex-01.c` but adds more XBraid features such as the residual feature, the user defined initial time-grid and full multigrid cycling.
- `examples/ex-01-expanded-bdf2.c`: same as ex-01-expanded.c, but uses BDF2 instead of backward Euler. This example makes use of the advanced shell-vector feature in order to implement BDF2.
- `examples/ex-01-expanded-f.f90`: same as ex-01-expanded.c, but implemented in f90.
- `examples/ex-01-refinement.c`: same as ex-01.c, but adds the refinement feature of XBraid. The refinement can be arbitrary or based on error estimate.

4.4 One-Dimensional Heat Equation

In this example, we assume familiarity with The Simplest Example. This example is a time-only parallel example that implements the 1D heat equation,

$$\delta/\delta_t \; u(x,t) = \Delta\, u(x,t) + g(x,t),$$

as opposed to The Simplest Example, which implements only a scalar ODE for one degree-of-freedom in space. There is no spatial parallelism, as a serial cyclic reduction algorithm is used to invert the tri-diagonal spatial operators. The space-time discretization is the standard 3-point finite difference stencil ($[-1, 2, -1]$), scaled by mesh widths. Backward Euler is used in time.

This example consists of three files and two executables.

- `examples/ex-02-serial.c`: This file compiles into its own executable `ex-02-serial` and represents a simple example user application that does sequential time-stepping. This file represents where a new XBraid user would start, in terms of converting a sequential time-stepping code to XBraid.
- `examples/ex-02.c`: This file compiles into its own executable `ex-02` and represents a time-parallel XBraid wrapping of the user application `ex-02-serial`.
- `ex-02-lib.c`: This file contains shared functions used by the time-serial version and the time-parallel version. This file provides the basic functionality of this problem. For instance, *take_step(u, tstart, tstop, ...)* carries out a step, moving the vector *u* from time *tstart* to time *tstop*.

4.5 Two-Dimensional Heat Equation

In this example, we assume familiarity with The Simplest Example and describe the major ways in which this example differs. This example is a full space-time parallel example, as opposed to The Simplest Example, which implements only a scalar ODE for one degree-of-freedom in space. We solve the heat equation in 2D,

$$\delta/\delta_t \; u(x,y,t) = \Delta\, u(x,y,t) + g(x,y,t).$$

For spatial parallelism, we rely on the `hypre` package where the SemiStruct interface is used to define our spatial discretization stencil and form our time stepping scheme, the backward Euler method. The spatial discretization is just the standard 5-point finite difference stencil ($[-1; -1, 4, -1; -1]$), scaled by mesh widths, and the PFMG solver is used for the solves required by backward Euler. Please see the hypre manual and examples for more information on the SemiStruct interface and PFMG. Although, the hypre specific calls have mostly been abstracted away for this example, and so it is not necessary to be familiar with the SemiStruct interface for this example.

This example consists of three files and two executables.

- examples/ex-03-serial.c: This file compiles into its own executable `ex-03-serial` and represents a simple example user application. This file supports only parallelism in space and represents a basic approach to doing efficient sequential time stepping with the backward Euler scheme. Note that the hypre solver used (PFMG) to carry out the time stepping is highly efficient.
- examples/ex-03.c: This file compiles into its own executable `ex-03` and represents a basic example of wrapping the user application `ex-03-serial`. We will go over the wrappers below.
- ex-03-lib.c: This file contains shared functions used by the time-serial version and the time-parallel version. This is where most of the hypre specific calls reside. This file provides the basic functionality of this problem. For instance, *take_step(u, tstart, tstop, ...)* carries out a step, moving the vector *u* from time *tstart* to time *tstop* and *setUpImplicitMatrix(...)* constructs the matrix to be inverted by PFMG for the backward Euler method.

User Defined Structures and Wrappers

We now discuss in more detail the important data structures and wrapper routines in `examples/ex-03.c`. The actual code for this example is quite simple and it is recommended to read through it after this overview.

The two data structures are:

1. **App**: This holds a wide variety of information and is *global* in that it is passed to every user function. This structure holds everything that the user will need to carry out a simulation. One important structure contained in the *app* is the *simulation_manager*. This is a structure native to the user code `ex-03-lib.c`. This structure conveniently holds the information needed by the user code to carry out a time step. For instance,

```
app->man->A
```

   is the time stepping matrix,

```
app->man->solver
```

   is the hypre PFMG solver object,

```
app->man->dt
```

   is the current time step size. The app is defined as

```
typedef struct _braid_App_struct {
   MPI_Comm             comm;             /* global communicator */
   MPI_Comm             comm_t;           /* communicator for parallelizing in time  */
   MPI_Comm             comm_x;           /* communicator for parallelizing in space  */
   int                  pt;               /* number of processors in time  */
   simulation_manager  *man;              /* user's simulation manager structure */
   HYPRE_SStructVector  e;                /* temporary vector used for error computations */
   int                  nA;               /* number of spatial matrices created */
   HYPRE_SStructMatrix *A;                /* array of spatial matrices, size nA, one per level*/
   double              *dt_A;             /* array of time step sizes, size nA, one per level*/
   HYPRE_StructSolver  *solver;           /* array of PFMG solvers, size nA, one per level*/
   int                  use_rand;         /* binary value, use random or zero initial guess */
   int                 *runtime_max_iter; /* runtime info for number of PFMG iterations*/
   int                 *max_iter_x;       /* maximum iteration limits for PFMG */
} my_App;
```

The app contains all the information needed to take a time step with the user code for an arbitrary time step size. See the *Step* function below for more detail.

1. **Vector**: this defines a state vector at a certain time value. Here, the vector is a structure containing a native hypre data-type, the *SStructVector*, which describes a vector over the spatial grid. Note that *my_Vector* is used to define *braid_Vector*.

```
typedef struct _braid_Vector_struct {
   HYPRE_SStructVector   x;
} my_Vector;
```

The user must also define a few wrapper routines. Note, that the `app` structure is the first argument to every function.

1. **Step**: This function tells XBraid how to take a time step, and is the core user routine. This function advances the vector *u* from time *tstart* to time *tstop*. A few important things to note are as follows.
   - The time values are given to the user through the *status* structure and associated *Get* routines.
   - The basic strategy is to see if a matrix and solver already exist for this dt value. If not, generate a new matrix and solver and store them in the *app* structure. If they do already exist, then re-use the data.
   - To carry out a step, the user routines from `ex-03-lib.c` rely on a few crucial data members *man->dt*, *man->A* and *man-solver*. We overwrite these members with the correct information for the time step size in question. Then, we pass *man* and *u* to the user function *take_step(...)* which evolves *u*.
   - The forcing term g_i is wrapped into the *take_step(...)* function. Thus, $\Phi(u_i) \rightarrow u_{i+1}$.

```
int my_Step(braid_App        app,
            braid_Vector     u,
            braid_StepStatus status)
{
   double tstart;             /* current time */
   double tstop;              /* evolve u to this time*/
   int i, A_idx;
   int iters_taken = -1;

   /* Grab status of current time step */
   braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

   /* Check matrix lookup table to see if this matrix already exists*/
   A_idx = -1.0;
   for( i = 0; i < app->nA; i++ ){
      if( fabs( app->dt_A[i] - (tstop-tstart) )/(tstop-tstart) < 1e-10) {
         A_idx = i;
         break;
      }
   }

   /* We need to "trick" the user's manager with the new dt */
   app->man->dt = tstop - tstart;

   /* Set up a new matrix and solver and store in app */
   if( A_idx == -1.0 ){
      A_idx = i;
      app->nA++;
      app->dt_A[A_idx] = tstop-tstart;

      setUpImplicitMatrix( app->man );
      app->A[A_idx] = app->man->A;

      setUpStructSolver( app->man, u->x, u->x );
      app->solver[A_idx] = app->man->solver;
   }

   /* Time integration to next time point: Solve the system Ax = b.
    * First, "trick" the user's manager with the right matrix and solver */
   app->man->A = app->A[A_idx];
   app->man->solver = app->solver[A_idx];
   ...
   /* Take step */
   take_step(app->man, u->x, tstart, tstop);
   ...
   return 0;
}
```

2. There are other functions, **Init**, **Clone**, **Free**, **Sum**, **SpatialNorm**, **Access**, **BufSize**, **BufPack** and **BufUnpack**, which also must be written. These functions are all simple for this example, as for the case of The Simplest Example. All we do here is standard operations on a spatial vector such as initialize, clone, take an inner-product, pack, etc... We refer the reader to `ex-03.c`.

Running XBraid for this Example

To initialize and run XBraid, the procedure is similar to The Simplest Example. Only here, we have to both initialize the user code and XBraid. The code that is specific to the user's application comes directly from the existing serial simulation code. If you compare `ex-03-serial.c` and `ex-03.c`, you will see that most of the code setting up the user's data structures and defining the wrapper functions are simply lifted from the serial simulation.

Taking excerpts from the function *main()* in ex-03.c, we first initialize the user's simulation manager with code like

```
...
app->man->px    = 1;   /* my processor number in the x-direction */
app->man->py    = 1;   /* my processor number in the y-direction */
```

```
                          /* px*py=num procs in space */
app->man->nx    = 17;  /* number of points in the x-dim */
app->man->ny    = 17;  /* number of points in the y-dim */
app->man->nt    = 32;  /* number of time steps */
...
```

We also define default XBraid parameters with code like

```
...
max_levels      = 15; /* Max levels for XBraid solver */
min_coarse      = 3;  /* Minimum possible coarse grid size */
nrelax          = 1;  /* Number of CF relaxation sweeps on all levels */
...
```

The XBraid app must also be initialized with code like

```
...
app->comm   = comm;
app->tstart = tstart;
app->tstop  = tstop;
app->ntime  = ntime;
```

Then, the data structure definitions and wrapper routines are passed to XBraid.

```
braid_Core  core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
        my_Step, my_Init, my_Clone, my_Free, my_Sum, my_SpatialNorm,
        my_Access, my_BufSize, my_BufPack, my_BufUnpack, &core);
```

Then, XBraid options are set with calls like

```
...
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetNRelax(core, -1, nrelax);
...
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run ex-03, type

```
ex-03 -help
```

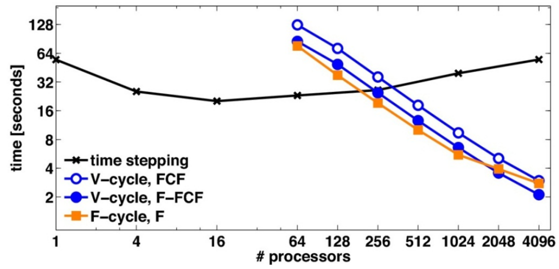
As a simple example, try the following.

```
mpirun -np 8 ex-03 -pgrid 2 2 2 -nt 256
```

4.5.1 Scaling Study with this Example

Here, we carry out a simple strong scaling study for this example. The "time stepping" data set represents sequential time stepping and was generated using `examples/ex-03-serial`. The time-parallel data set was generated using `examples/ex-03`. The problem setup is as follows.

- Backwards Euler is used as the time stepper. This is the only time stepper supported by `ex-03`.



- We used a Linux cluster with 4 cores per node, a Sandybridge Intel chipset, and a fast Infiniband interconnect.
- The space-time problem size was $129^2 \times 16,192$ over the unit cube $[0,1] \times [0,1] \times [0,1]$.
- The coarsening factor was $m = 16$ on the finest level and $m = 2$ on coarser levels.
- Since 16 processors optimized the serial time stepping approach, 16 processors in space are also used for the XBraid experiments. So for instance 512 processrs in the plot corresponds to 16 processors in space and 32 processors in time, $16 * 32 = 512$. Thus, each processor owns a space-time hypercube of $(129^2/16) \times (16,192/32)$. See Parallel decomposition and memory for a depiction of how XBraid breaks the problem up.
- Various relaxation and V and F cycling strategies are experimented with.
  - *V-cycle, FCF* denotes V-cycles and FCF-relaxation on each level.
  - *V-cycle, F-FCF* denotes V-cycles and F-relaxation on the finest level and FCF-relaxation on all coarser levels.
  - *F-cycle, F* denotes F-cycles and F-relaxation on each level.
- The initial guess at time values for $t > 0$ is zero, which is typical.
- The halting tolerance corresponds to a discrete L2-norm and was

$$\text{tol} = \frac{10^{-8}}{\sqrt{(h_x)^2 h_t}},$$

  where h_x and h_t are the spatial and temporal grid spacings, respectively.

  This corresponds to passing *tol* to braid_SetAbsTol, passing *2* to braid_SetTemporalNorm and defining braid_PtFcnSpatialNorm to be the standard Euclidean 2-norm. All together, this appropriately scales the space-time residual in way that is relative to the number of space-time grid points (i.e., it approximates the L2-norm).

To re-run this scaling study, a sample run string for ex-03 is

```
mpirun -np 64 ex-03 -pgrid 4 4 4 -nx 129 129 -nt 16129 -cf0 16 -cf 2 -nu 1 -use_rand 0
```

To re-run the baseline sequential time stepper, ex-03-serial, try

```
mpirun -np 64 ex-03-serial -pgrid 8 8 -nx 129 129 -nt 16129
```

For explanations of the command line parameters, type

```
ex-03-serial -help
ex-03 -help
```

Regarding the performance, we can say

- The best speedup is 10x and this would grow if more processors were available.

- Although not shown, the iteration counts here are about 10-15 XBraid iterations. See `Parallel Time Integration with Multigrid` for the exact iteration counts.
- At smaller core counts, serial time stepping is faster. But at about 256 processors, there is a crossover and XBraid is faster.
- You can see the impact of the cycling and relaxation strategies discussed in Cycling and relaxation strategies. For instance, even though *V-cycle, F-FCF* is a weaker relaxation strategy than *V-cycle, FCF* (i.e., the XBraid convergence is slower), *V-cycle, F-FCF* has a faster time to solution than *V-cycle, FCF* because each cycle is cheaper.
- In general, one level of aggressive coarsening (here by a factor 16) followed by slower coarsening was found to be best on this machine.

Achieving the best speedup can require some tuning, and it is recommended to read `Parallel Time Integration with Multigrid` where this 2D heat equation example is explored in much more detail.

4.6 Simplest XBraid_Adjoint example

The file `examples/ex-01-adjoint.c` extends the simple scalar ODE example in `ex-01.c` for computing adjoint-based sensitivities. See The Simplest Example. The scalar ODE is

$$u_t(t) = \lambda u(t) \quad \forall t \in (0,T),$$

where λ is considered the design variable. We consider an objective function of the form

$$J(u,\lambda) = \int_0^T \frac{1}{T}\|u(t)\|^2 dt.$$

User Defined Structures and Wrappers

The two user-defined data structures are:

1. **Vector:** This structure is unchanged from The Simplest Example, and contains a single scalar representing the state at a given time.

```
typedef struct _braid_Vector_struct
{
   double value;
} my_Vector;
```

2. **App:** This structure holds two additional elements when compared to The Simplest Example : the *design* and the *reduced gradient*. This ensures that both are accessible in all user routines.

```
typedef struct _braid_App_struct
{
   int       rank;
   double    design;
   double    gradient;
} my_App;
```

The user must also define a few *additional* wrapper routines. Note, that the app structure continues to be the first argument to every function.

1. All user-defined routines from `examples/ex-01.c` stay the same, except `Step()`, which must be changed to account for the new design parameter in `app`.

2. The user's **Step** routine queries the `app` to get the design and propagates the `braid_Vector u` forward in time for one time step:

```
int
my_Step(braid_App        app,
        braid_Vector     ustop,
        braid_Vector     fstop,
        braid_Vector     u,
        braid_StepStatus status)
{
   double tstart;             /* current time */
   double tstop;              /* evolve to this time*/
   braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

   /* Get the design variable from the app */
   double lambda = app->design;

   /* Use backward Euler to propagate the solution */
   (u->value) = 1./(1. - lambda * (tstop-tstart))*(u->value);

   return 0;
}
```

3. **ObjectiveT**: This new routine evaluates the time-dependent part of the objective function at a local time t_i, i.e. it returns the integrand $f(u_i, \lambda) = \frac{1}{T}\|u_i\|_2^2$.

```
int
my_ObjectiveT(braid_App              app,
              braid_Vector           u,
              braid_ObjectiveStatus  ostatus,
              double                *objectiveT_ptr)
{
   /* Get the total number of time steps */
   braid_ObjectiveStatusGetNTPoints(ostatus, &ntime);

   /* Evaluate the local objective: 1/N u(t)^2 */
   objT = 1. / ntime * (u->value) * (u->value);

   *objectiveT_ptr = objT;
   return 0;
}
```

The `ObjectiveStatus` can be queried for information about the current status of XBraid (e.g., what is the current time value, time-index, number of time steps, current iteration number, etc...).

XBraid_Adjoint calls the `ObjectiveT` function on the finest time-grid level during the down-cycle of the multigrid algorithm and adds the value to a global objective function value with a simple summation. Thus, any user-specific integration formula of the objective function must be here.

4. **ObjectiveT_diff**: This new routine updates the adjoint variable `u_bar` and the reduced gradient with the transposed partial derivatives of `ObjectiveT` multiplied by the scalar input $\bar{F}$, i.e.,

$$\bar{u}_i = \frac{\partial f(u_i, \lambda)}{\partial u_i}^T \bar{F} \quad \text{and} \quad \bar{\rho} += \frac{\partial f(u_i, \lambda)}{\partial \rho}^T \bar{F}.$$

Note that $\bar{u}_i$ gets overwritten (" $=$"), whereas ρ is updated (" $+=$").

```
int
my_ObjectiveT_diff(braid_App             app,
                   braid_Vector          u,
                   braid_Vector          u_bar,
                   braid_Real            F_bar,
                   braid_ObjectiveStatus ostatus)
{
   int    ntime;
```

```
    double ddu;      /* Derivative wrt u */
    double ddesign;  /* Derivative wrt design */

    /* Get the total number of time steps */
    braid_ObjectiveStatusGetNTPoints(ostatus, &ntime);

    /* Partial derivative with respect to u times F_bar */
    ddu = 2. / ntime * u->value * F_bar;

    /* Partial derivative with respect to design times F_bar*/
    ddesign = 0.0 * F_bar;

    /* Update u_bar and gradient */
    u_bar->value   = ddu;
    app->gradient += ddesign;

    return 0;
}
```

5. **Step_diff**: This new routine computes transposed partial derivatives of the `Step` routine multiplied with the adjoint vector `u_bar` ($\bar{u}_i$), i.e.,

$$\bar{u}_i = \left(\frac{\partial \Phi_{i+1}(u_i,\rho)}{\partial u_i}\right)^T \bar{u}_i \quad \text{and} \quad \bar{\rho} + = \left(\frac{\partial \Phi_{i+1}(u_i,\rho)}{\partial \rho}\right)^T \bar{u}_i.$$

```
int
my_Step_diff(braid_App           app,
             braid_Vector        ustop,
             braid_Vector        u,
             braid_Vector        ustop_bar,
             braid_Vector        u_bar,
             braid_StepStatus    status)
{
   double ddu;      /* Derivative wrt u */
   double ddesign;  /* Derivative wrt design */

   /* Get the time step size */
   double tstop, tstart, deltat;
   braid_StepStatusGetTstartTstop(status, &tstart, &tstop);
   deltat = tstop - tstart;

   /* Get the design from the app */
   double lambda = app->design;

   /* Transposed derivative of step wrt u times u_bar */
   ddu = 1./(1. - lambda * deltat) * (u_bar->value);

   /* Transposed derivative of step wrt design times u_bar */
   ddesign = (deltat * (u->value)) / pow(1. - deltat*lambda,2) * (u_bar->value);

   /* Update u_bar and gradient */
   u_bar->value      = ddu;
   app->gradient    += ddesign;

   return 0;
}
```

Important note on the usage of ustop: If the `Step` routine uses the input vector `ustop` instead of `u` (typically for initializing a (non-)linear solve within Φ), then `Step_diff` must update `ustop_bar` instead of `u_bar` and set `u_bar` to zero:

$$\overline{ustop} + = \left(\frac{\partial \Phi_{i+1}(ustop,\rho)}{\partial\, ustop}\right)^T \bar{u}_i \quad \text{and} \quad \bar{u}_i = 0.0.$$

6. **ResetGradient**: This new routine sets the gradient to zero.

```
int
my_ResetGradient(braid_App app)
{
   app->gradient = 0.0;
   return 0;
}
```

XBraid_Adjoint calls this routine before each iteration such that old gradient information is removed properly.

Running XBraid_Adjoint for this example

The workflow for computing adjoint sensitivities with XBraid_Adjoint alongside the primal state computation closely follows XBraid's workflow. The user's *main* file will first set up the `app` structure, holding the additional information on an initial design and zero gradient. Then, all the setup calls done in Running XBraid for the Simplest Example will also be done.

The XBraid_Adjoint specific calls are as follows. After `braid_Init(...)` is called, the user initializes XBraid_Adjoint by calling

```
/* Initialize XBraid_Adjoint */
braid_InitAdjoint( my_ObjectiveT, my_ObjectiveT_diff, my_Step_diff, my_ResetGradient, &core);
```

Next, in addition to the usual XBraid options for controlling the multigrid iterations, the adjoint solver's accuracy is set by calling

```
braid_SetAbsTolAdjoint(core, 1e-6);
```

After that, one call to

```
/* Run simulation and adjoint-based gradient computation */
braid_Drive(core);
```

runs the multigrid iterations with additional adjoint sensitivity computations (i.e. the piggy-back iterations). After it finishes, the objective function value can be accessed by calling

```
/* Get the objective function value from XBraid */
braid_GetObjective(core, &objective);
```

Further, the reduced gradient, which is stored in the user's `App` structure, holds the sensitivity information $dJ/d\rho$. As this information is local to all the time-processors, the user is responsible for summing up the gradients from all time-processors, if necessary. This usually involves an `MPI_Allreduce` call as in

```
/* Collect sensitivities from all processors */
double mygradient = app->gradient;
MPI_Allreduce(&mygradient, &(app->gradient), 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Lastly, the gradient computed with XBraid_Adjoint is verified using Finite Differences. See the source code `examples/ex-01-adjoint.c` for details.

4.7 Optimization with the Simplest Example

The file `examples/ex-01-optimization.c` implements a simple optimization iteration by extending `examples/ex-01-adjoint.c`, described in Simplest XBraid_Adjoint example. This example solves an inverse design problem for the simple scalar ODE example:

$$\min \ \frac{1}{2}\left(\int_0^T \frac{1}{T}\|u(t)\|^2 dt - J_{\text{Target}}\right)^2 + \frac{\gamma}{2}\|\lambda\|^2$$

$$\text{s.t.} \quad \frac{\partial}{\partial t}u(t) = \lambda u(t) \quad \forall t \in (0,T)$$

where J_{Target} is a fixed and precomputed target value and $\gamma > 0$ is a fixed relaxation parameter. Those fixed values are stored within the `App`.

User Defined Structures and Wrappers

In order to evaluate the time-independent part of the objective function (e.g. the postprocessing function F) and its derivative, two additional user routines are necessary. *There are no new user-defined data structures.*

1. **PostprocessObjective**: This function evaluates the tracking-type objective function and the regularization term. The input variable `integral` contains the integral-part of the objective and returns the objective that is to be minimized $F(I)$:

```
/* Evaluate the time-independent part of the objective function */
int
my_PostprocessObjective(braid_App  app,
                     double        integral,
                     double       *postprocess
                     )
{
   double F;

   /* Tracking-type functional */
   F  = 1./2. * pow(integral - app->target,2);

   /* Regularization term */
   F += (app->gamma) / 2. * pow(app->design,2);

   *postprocess = F;
    return 0;
}
```

1. **PostprocessObjective_diff**: This provides XBraid_Adjoint with the partial derivatives of the `Postprocess↵ Objective` routine, i.e.

$$\bar{F} = \frac{\partial F(I,\lambda)}{\partial I} \quad \text{and} \quad \bar{\rho} + = \frac{\partial F(I,\lambda)}{\partial \lambda}$$

```
int
my_PostprocessObjective_diff(braid_App   app,
                             double      integral,
                             double     *F_bar
                             )
{

   /* Derivative of tracking type function */
   *F_bar = integral - app->target;

   /* Derivative of regularization term */
   app->gradient += (app->gamma) * (app->design);
   return 0;
}
```

These routines are optional for XBraid_Adjoint. Therefore, they need to be passed to XBraid_Adjoint after the initialization with `braid_Init(...)` and `braid_InitAdjoint(...)` in the user's *main* file:

```
/* Optional: Set the tracking type objective function and derivative */
braid_SetPostprocessObjective(core, my_PostprocessObjective);
braid_SetPostprocessObjective_diff(core, my_PostprocessObjective_diff);
```

Running an Optimization Cycle with XBraid_Adjoint

XBraid_Adjoint does not natively implement any optimization algorithms. Instead, we provide examples showing how one can easily use XBraid_Adjoint inside an optimization cycle. Here, one iteration of the optimization cycle consists of the following steps:

1. First, we run XBraid_Adjoint to solve the primal and adjoint dynamics:

   ```
   braid_Drive(core);
   ```

2. Get the value of the objective function with

   ```
   braid_GetObjective(core, &objective);
   ```

3. Gradient information is stored in the `app` structure. Since it is local to all temporal processors, we need to invoke an `MPI_Allreduce` call which sums up the local sensitivities:

   ```
   mygradient = app->gradient;
   MPI_Allreduce(&mygradient, &app->gradient, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   ```

   Note: For time-dependent design variables, summing over all processors might not be necessary, since information is needed only locally in time. See `examples/ex-04.c` for a time-dependent design example.

4. Update the design variable using the gradient information. Here, we implement a simple steepest descent update into the direction of the negative gradient:

   ```
   app->design -= stepsize * app->gradient;
   ```

   Here, a fixed step size is used to update the design variable. Usually, a line-search procedure should be implemented in order to find a suitable step length that minimizes the objective function along the update direction. However to carry out a line search, we must re-evaluate the objective function for different design value(s). Thus, the option braid_SetObjectiveOnly(core, 1) can be used. After this option has been set, any further call to `braid_Drive(core)` will then only run a primal XBraid simulation and carry out an objective function evaluation. No gradients will be computed, which saves computational time. After the line search, make sure to reset XBraid_Adjoint for gradient computation with `braid_SetObjectiveOnly(core, 0)`.

5. The optimization iterations are stopped when the norm of the gradient is below a prescribed tolerance.

4.8 A Simple Optimal Control Problem

This example demonstrates the use of XBraid_Adjoint for solving an optimal control problem with time-dependent design variables:

$$\min \int_0^1 u_1(t)^2 + u_2(t)^2 + \gamma c(t)^2 \, dt$$

$$\text{s.t.} \quad \begin{array}{ll} \frac{\partial}{\partial t} u_1(t) = u_2(t) & \forall t \in (0,1) \\ \frac{\partial}{\partial t} u_2(t) = -u_2(t) + c(t) & \forall t \in (0,1) \end{array}$$

with initial condition $u_1(0) = 0, u_2(0) = -1$ and piecewise constant control (design) variable $c(t)$.

The example consists of three files, meant to indicate how one can take a time-serial implementation for an optimal control problem and create a corresponding XBraid_Adjoint implementation.

- `examples/ex-04-serial.c`: Compiles into its own executable `examples/ex-04-serial`, which solves the optimal control problem using time-serial forward-propagation of state variables and time-serial backward-propagation of the adjoint variables in each iteration of an outer optimization cycle.
- `examples/ex-04.c`: Compiles into `ex-04`. This solves the same optimization problem in time-parallel by replacing the forward- and backward-propagation of state and adjoint by the time-parallel XBraid and XBraid_↩ Adjoint solvers.

- `examples/ex-04-lib.c`: Contains the routines that are shared by both the serial and the time-parallel implementation. Study this file, and discover that most of the important code setting up the user-defined data structures and wrapper routines are simply lifted from the serial simulation.

4.9 Running and Testing XBraid

The best overall test for XBraid, is to set the maximum number of levels to 1 (see braid_SetMaxLevels) which will carry out a sequential time stepping test. Take the output given to you by your *Access* function and compare it to output from a non-XBraid run. Is everything OK? Once this is complete, repeat for multilevel XBraid, and check that the solution is correct (that is, it matches a serial run to within tolerance).

At a lower level, to do sanity checks of your data structures and wrapper routines, there are also XBraid test functions, which can be easily run. The test routines also take as arguments the *app* structure, spatial communicator *comm_x*, a stream like *stdout* for test output and a time step size *dt* to test. After these arguments, function pointers to wrapper routines are the rest of the arguments. Some of the tests can return a boolean variable to indicate correctness.

```
/* Test init(), access(), free() */
braid_TestInitAccess( app, comm_x, stdout, dt, my_Init, my_Access, my_Free);

/* Test clone() */
braid_TestClone( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone);

/* Test sum() */
braid_TestSum( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone, my_Sum);

/* Test spatialnorm() */
correct = braid_TestSpatialNorm( app, comm_x, stdout, dt, my_Init, my_Free, my_Clone,
                          my_Sum, my_SpatialNorm);

/* Test bufsize(), bufpack(), bufunpack() */
correct = braid_TestBuf( app, comm_x, stdout, dt, my_Init, my_Free, my_Sum, my_SpatialNorm,
                    my_BufSize, my_BufPack, my_BufUnpack);

/* Test coarsen and refine */
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                       my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                       my_CoarsenInjection, my_Refine);
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                      my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                      my_CoarsenBilinear, my_Refine);
```

4.10 Fortan90 Interface, C++ Interface, Python Interface, and More Complicated Examples

We have Fortran90, C++, and Python interfaces. For Fortran 90, see `examples/ex-01f.f90`. For C++ see `braid.hpp` and `examples/ex-01-pp.cpp` For more complicated C++ examples, see the various C++ examples in `drivers/drive-**.cpp`. For Python, see the directories `examples/ex-01-cython` and `examples/ex-01-cython-alt`.

For a discussion of more complex problems please see our project `publications website` for our recent publications concerning some of these varied applications.

5 Examples: compiling and running

For C/C++/Fortran examples, type

```
ex-* -help
```

for instructions on how to run. To run the C/C++/Fortran examples, type

```
mpirun -np 4 ex-*  [args]
```

For the Cython examples, see the corresponding ∗.pyx file.

1. ex-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies. See Section (The Simplest Example) for more discussion of this example. There are seven versions of this example,
   - *ex-01.c*: simplest possible implementation, start reading this example first
   - *ex-01-expanded.c*: same as ex-01.c but adds more XBraid features
   - *ex-01-expanded-bdf2.c*: same as ex-01-expanded.c, but uses BDF2 instead of backward Euler
   - *ex-01-expanded-f.f90*: same as ex-01-expanded.c, but implemented in f90
   - *ex-01-refinement.c*: same as ex-01.c, but adds the refinement feature
   - *ex-01-adjoint.c*: adds adjoint-based gradient computation to ex-01.c
   - *ex-01-optimization.c*: gradient-based optimization cycle for ex-01-c
   - *ex-01-cython/*: is a directory containing an example using the Braid-Cython interface defined in braid.↩ pyx (braid/braid.pyx). It solves the same scalar ODE equation as the ex-01 series described above. This example uses a Python-like syntax, in contrast to the ex-01-cython-alt example, which uses a C-style syntax. For instructions on running and compiling, see

     ```
     examples/ex-01-cython/ex_01.pyx
     ```

     and

     ```
     examples/ex-01-cython/ex_01-setup.py
     ```

   - *ex-01-cython-alt/*: is a directory containing another example using the Braid-Cython interface defined in braid.pyx (braid/braid.pyx). It solves the same scalar ODE equation as the ex-01 series described above. This example uses a lower-level C-like syntax for most of it's code, in contrast to the ex-01-cython example, which uses a Python-style syntax.
     For instructions on running and compiling, see

     ```
     examples/ex-01-cython-alt/ex_01_alt.pyx
     ```

     and

     ```
     examples/ex-01-cython-alt/ex_01_alt-setup.py
     ```

2. ex-02 implements the 1D heat equation on a regular grid, using a very simple implementation. This is the next example to read after the various ex-01 cases.
3. ex-03 implements the 2D heat equation on a regular grid. You must have `hypre` installed and these variables in examples/Makefile set correctly

   ```
   HYPRE_DIR = ../../linear_solvers/hypre
   HYPRE_FLAGS = -I$(HYPRE_DIR)/include
   HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
   ```

   Only implicit time stepping (backward Euler) is supported. See Section (Two-Dimensional Heat Equation) for more discussion of this example. The driver

   ```
   drivers/drive-diffusion
   ```

   is a more sophisticated version of this simple example that supports explicit time stepping and spatial coarsening.
4. ex-04 solves a simple optimal control problem with time-dependent design variable using a simple steepest-descent optimization iteration.
5. Directory ex-05-cython/ solves a simple 1D heat equation using the Cython interface

   ```
   examples/ex-05-cython/ex_05.pyx
   ```

and

```
examples/ex-05-cython/ex_05-setup.py
```

6. ex-06 solve a simple scalar ODE, but allows for use of the built-in Richardson-based error estimator and accuracy improving extrapolation. With the "-refinet" option, the error estimator allows for adaptive refinement in time, and with the "-richardson" option, Richardson extrapolation is used improve the solution at fine-level C-points.

   The viz script,

   ```
   examples/viz-ex-06.py
   ```

   allows you to visualize the solution, error, and error estimate. The use of "-richardson" notably improves the accuracy of the solution.

   The Richardson-based error estimates and/or extrapolation are only available after the first Braid iteration, in that the coarse level solution must be available to compute the error estimate and extrapolation. Thus, after an adaptive refinement (and new hierarchy is constructed), another iteration is again required for the error estimate to be available. If the error estimate isn't available, Braid returns a value of -1. See this example and the comments therein for more details.

6 Drivers: compiling and running

Type

```
drive-* -help
```

for instructions on how to run any driver.

To run the examples, type

```
mpirun -np 4 drive-*  [args]
```

1. *drive-diffusion-2D* implements the 2D heat equation on a regular grid. You must have hypre installed and these variables in examples/Makefile set correctly

   ```
   HYPRE_DIR = ../../linear_solvers/hypre
   HYPRE_FLAGS = -I$(HYPRE_DIR)/include
   HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
   ```

   This driver also support spatial coarsening and explicit time stepping. This allows you to use explicit time stepping on each Braid level, regardless of time step size.

2. *drive-burgers-1D* implements Burger's equation (and also linear advection) in 1D using forward or backward Euler in time and Lax-Friedrichs in space. Spatial coarsening is supported, allowing for stable time stepping on coarse time-grids.

   See also *viz-burgers.py* for visualizing the output.

3. *drive-lorenz* implements the Lorenz equation, with it's trademark attractors. This problem has not been researched very extensively, and XBraid's behavior is not yet well understood. Convergence stagnates, but is the solution "good enough" from a statistical point-of-view?

   See also *viz-lorenz.py* for visualizing the output.

4. *drive-diffusion* is a sophisticated test bed for finite element discretizations of the heat equation. It relies on the mfem package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.

- Unpack and install Metis
- Unpack and install hypre
- Unpack mfem. Then make sure to set these variables correctly in the mfem Makefile:

```
USE_METIS_5 = YES
HYPRE_DIR   = where_ever_linear_solvers_is/hypre
```

- Make the parallel version of mfem first by typing

```
make parallel
```

- Make GLVIS. Set these variables in the glvis makefile

```
MFEM_DIR   = mfem_location
MFEM_LIB   = -L$(MFEM_DIR) -lmfem
```

- Go to braid/examples and set these Makefile variables,

```
METIS_DIR = ../../metis-5.1.0/lib
MFEM_DIR = ../../mfem
MFEM_FLAGS = -I$(MFEM_DIR)
MFEM_LIB = -L$(MFEM_DIR) -lmfem -L$(METIS_DIR) -lmetis
```

  then type

```
make drive-diffusion
```

- To run drive-diffusion and glvis, open two windows. In one, start a glvis session

```
./glvis
```

  Then, in the other window, run drive-diffusion

```
mpirun -np ... drive-diffusion [args]
```

  Glvis will listen on a port to which drive-diffusion will dump visualization information.

5. The other drive-.cpp files use MFEM to implement other PDEs

   - *drive-adv-diff-DG*: implements advection(-diffusion) with a discontinuous Galerkin discretization. This driver is under developement.
   - *drive-diffusion-1D-moving-mesh*: implements the 1D heat equation, but with a moving mesh that adapts to the forcing function so that the mesh equidistributes the arc-length of the solution.
   - *drive-diffusion-1D-moving-mesh-serial*: implements a serial time-stepping version of the above problem.
   - *drive-pLaplacian*: implements the 2D the p-Laplacian (nonlinear diffusion).
   - *drive-diffusion-ben*: implements the 2D/3D diffusion equation with time-dependent coefficients. This is essentially equivalent to drive-diffusion, and could be removed, but we're keeping it around because it implements linear diffusion in the same way that the p-Laplacian driver implemented nonlinear diffusion. This makes it suitable for head-to-head timings.
   - *drive-lin-elasticity*: implements time-dependent linearized elasticity and is under development.
   - *drive-nonlin-elasticity*: implements time-dependent nonlinear elasticity and is under development.

6. Directory drive-adv-diff-1D-Cython/ solves a simple 1D advection-diffussion equation using the Cython interface and numerous spatial and temporal discretizations

```
drivers/drive-adv-diff-1D-Cython/drive_adv_diff_1D.pyx
```

   and

```
drivers/drive-adv-diff-1D-Cython/drive_adv_diff_1D-setup.py
```

7 File naming conventions

These are the general filenaming conventions for Braid

User interface routines in braid begin with `braid_` and all other internal non-user routines begin with `_braid_`. This helps to prevent name clashes when working with other libraries and helps to clearly distinguish user routines that are supported and maintained.

To keep things somewhat organized, all user header files and implementation files should have names that begin with `braid`, for example, `braid.h`, `braid.c`, `braid_status.c`, ... There should be no user interface prototypes or implementations that appear elsewhere.

Note that it is okay to include internal prototypes and implementations in these user interface files when it makes sense (say, as supporting routines), but this should generally be avoided.

An attempt has been made to simplify header file usage as much as possible by requiring only one header file for users, `braid.h`, and one header file for developers, `_braid.h`.

8 Module Index

8.1 Modules

Here is a list of all modules:

9 File Index

9.1 File List

Here is a list of all files with brief descriptions:

10 Module Documentation

10.1 Fortran 90 interface options

Macros

- #define braid_FMANGLE 1
- #define braid_Fortran_SpatialCoarsen 0
- #define braid_Fortran_Residual 1
- #define braid_Fortran_TimeGrid 1
- #define braid_Fortran_Sync 1

10.1.1 Detailed Description

Allows user to manually, at compile-time, turn on Fortran 90 interface options

10.1.2 Macro Definition Documentation

10.1.2.1 braid_FMANGLE

```
#define braid_FMANGLE 1
```

Define Fortran name-mangling schema, there are four supported options, see braid_F90_iface.c

10.1.2.2 braid_Fortran_Residual

```
#define braid_Fortran_Residual 1
```

Turn on the optional user-defined residual function

10.1.2.3 braid_Fortran_SpatialCoarsen

```
#define braid_Fortran_SpatialCoarsen 0
```

Turn on the optional user-defined spatial coarsening and refinement functions

10.1.2.4 braid_Fortran_Sync

```
#define braid_Fortran_Sync 1
```

Turn on the optional user-defined sync function

10.1.2.5 braid_Fortran_TimeGrid

```
#define braid_Fortran_TimeGrid 1
```

Turn on the optional user-defined time-grid function

10.2 Error Codes

Macros

- #define braid_INVALID_RNORM -1
- #define braid_ERROR_GENERIC 1 /* generic error */
- #define braid_ERROR_MEMORY 2 /* unable to allocate memory */
- #define braid_ERROR_ARG 4 /* argument error */

10.2.1 Detailed Description

10.2.2 Macro Definition Documentation

10.2.2.1 braid_ERROR_ARG

```
#define braid_ERROR_ARG 4 /* argument error */
```

10.2.2.2 braid_ERROR_GENERIC

```
#define braid_ERROR_GENERIC 1 /* generic error */
```

10.2.2.3 braid_ERROR_MEMORY

```
#define braid_ERROR_MEMORY 2 /* unable to allocate memory */
```

10.2.2.4 braid_INVALID_RNORM

```
#define braid_INVALID_RNORM -1
```

Value used to represent an invalid residual norm

10.3 User-written routines

Modules

- User-written routines for XBraid_Adjoint

Typedefs

- typedef struct _braid_App_struct ∗ braid_App
- typedef struct _braid_Vector_struct ∗ braid_Vector
- typedef braid_Int(∗ braid_PtFcnStep) (braid_App app, braid_Vector ustop, braid_Vector fstop, braid_Vector u, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnInit) (braid_App app, braid_Real t, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnClone) (braid_App app, braid_Vector u, braid_Vector ∗v_ptr)
- typedef braid_Int(∗ braid_PtFcnFree) (braid_App app, braid_Vector u)
- typedef braid_Int(∗ braid_PtFcnSum) (braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid_Vector y)
- typedef braid_Int(∗ braid_PtFcnSpatialNorm) (braid_App app, braid_Vector u, braid_Real ∗norm_ptr)
- typedef braid_Int(∗ braid_PtFcnAccess) (braid_App app, braid_Vector u, braid_AccessStatus status)
- typedef braid_Int(∗ braid_PtFcnSync) (braid_App app, braid_SyncStatus status)
- typedef braid_Int(∗ braid_PtFcnBufSize) (braid_App app, braid_Int ∗size_ptr, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnBufPack) (braid_App app, braid_Vector u, void ∗buffer, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnBufUnpack) (braid_App app, void ∗buffer, braid_Vector ∗u_ptr, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnResidual) (braid_App app, braid_Vector ustop, braid_Vector r, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnSCoarsen) (braid_App app, braid_Vector fu, braid_Vector ∗cu_ptr, braid_CoarsenRefStatus status)
- typedef braid_Int(∗ braid_PtFcnSRefine) (braid_App app, braid_Vector cu, braid_Vector ∗fu_ptr, braid_CoarsenRefStatus status)
- typedef braid_Int(∗ braid_PtFcnSInit) (braid_App app, braid_Real t, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnSClone) (braid_App app, braid_Vector u, braid_Vector ∗v_ptr)
- typedef braid_Int(∗ braid_PtFcnSFree) (braid_App app, braid_Vector u)
- typedef braid_Int(∗ braid_PtFcnTimeGrid) (braid_App app, braid_Real ∗ta, braid_Int ∗ilower, braid_Int ∗iupper)

10.3.1 Detailed Description

These are all the user-written data structures and routines. There are two data structures (braid_App and braid_Vector) for the user to define. And, there are a variety of function interfaces (defined through function pointer declarations) that the user must implement.

10.3.2 Typedef Documentation

10.3.2.1 braid_App

```
typedef struct _braid_App_struct* braid_App
```

This holds a wide variety of information and is `global` in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. For a simple example, this could just hold the global MPI communicator and a few values describing the temporal domain.

10.3.2.2 braid_PtFcnAccess

```
typedef braid_Int(* braid_PtFcnAccess) (braid_App app, braid_Vector u, braid_AccessStatus status)
```

Gives user access to XBraid and to the current vector *u* at time *t*. Most commonly, this lets the user write the vector to screen, file, etc... The user decides what is appropriate. Note how you are told the time value *t* of the vector *u* and other information in *status*. This lets you tailor the output, e.g., for only certain time values at certain XBraid iterations. Querrying status for such information is done through *braid_AccessStatusGet**(..)* routines.

The frequency of XBraid's calls to *access* is controlled through braid_SetAccessLevel. For instance, if access_level is set to 3, then *access* is called every XBraid iteration and on every XBraid level. In this case, querrying *status* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation.

Eventually, access will be broadened to allow the user to steer XBraid.

10.3.2.3 braid_PtFcnBufPack

```
typedef braid_Int(* braid_PtFcnBufPack) (braid_App app, braid_Vector u, void *buffer, braid_BufferStatus status)
```

This allows XBraid to send messages containing braid_Vectors. This routine packs a vector *u* into a *void * buffer* for MPI. The status structure holds information regarding the message. This is accessed through the *braid_BufferStatusGet**(..)* routines. Optionally, the user can set the message size through the status structure.

10.3.2.4 braid_PtFcnBufSize

```
typedef braid_Int(* braid_PtFcnBufSize) (braid_App app, braid_Int *size_ptr, braid_BufferStatus status)
```

This routine tells XBraid message sizes by computing an upper bound in bytes for an arbitrary braid_Vector. This size must be an upper bound for what BufPack and BufUnPack will assume.

10.3.2.5 braid_PtFcnBufUnpack

```
typedef braid_Int(* braid_PtFcnBufUnpack) (braid_App app, void *buffer, braid_Vector *u_ptr, braid_BufferStatus status)
```

This allows XBraid to receive messages containing braid_Vectors. This routine unpacks a *void * buffer* from MPI into a braid_Vector. The status structure, contains information conveying the type of message inside the buffer. This can be accessed through the *braid_BufferStatusGet**(..)* routines.

10.3.2.6 braid_PtFcnClone

```
typedef braid_Int(* braid_PtFcnClone) (braid_App app, braid_Vector u, braid_Vector *v_ptr)
```

Clone *u* into *v_ptr*

10.3.2.7 braid_PtFcnFree

```
typedef braid_Int(* braid_PtFcnFree) (braid_App app, braid_Vector u)
```

Free and deallocate *u*

10.3.2.8 braid_PtFcnInit

```
typedef braid_Int(* braid_PtFcnInit) (braid_App app, braid_Real t, braid_Vector *u_ptr)
```

Initializes a vector *u_ptr* at time *t*

10.3.2.9 braid_PtFcnResidual

```
typedef braid_Int(* braid_PtFcnResidual) (braid_App app, braid_Vector ustop, braid_Vector r, braid_StepStatus status)
```

This function (optional) computes the residual *r* at time *tstop*. On input, *r* holds the value of *u* at *tstart*, and *ustop* is the value of *u* at *tstop*. If used, set with braid_SetResidual.

Query the status structure with *braid_StepStatusGetTstart(status, &tstart)* and *braid_StepStatusGetTstop(status, &tstop)* to get *tstart* and *tstop*.

10.3.2.10 braid_PtFcnSClone

```
typedef braid_Int(* braid_PtFcnSClone) (braid_App app, braid_Vector u, braid_Vector *v_ptr)
```

Shell clone (optional)

10.3.2.11 braid_PtFcnSCoarsen

```
typedef braid_Int(* braid_PtFcnSCoarsen) (braid_App app, braid_Vector fu, braid_Vector *cu_ptr, braid_CoarsenRefStatus status)
```

Spatial coarsening (optional). Allows the user to coarsen when going from a fine time grid to a coarse time grid. This function is called on every vector at each level, thus you can coarsen the entire space time domain. The action of this function should match the braid_PtFcnSRefine function.

The user should query the status structure at run time with *braid_CoarsenRefGet**()* calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

10.3.2.12 braid_PtFcnSFree

```
typedef braid_Int(* braid_PtFcnSFree) (braid_App app, braid_Vector u)
```

Free the data of *u*, keep its shell (optional)

10.3.2.13 braid_PtFcnSInit

```
typedef braid_Int(* braid_PtFcnSInit) (braid_App app, braid_Real t, braid_Vector *u_ptr)
```

Shell initialization (optional)

10.3.2.14 braid_PtFcnSpatialNorm

```
typedef braid_Int(* braid_PtFcnSpatialNorm) (braid_App app, braid_Vector u, braid_Real *norm_ptr)
```

Carry out a spatial norm by taking the norm of a braid_Vector *norm_ptr* = $||$ *u* $||$ A common choice is the standard Eucliden norm, but many other choices are possible, such as an L2-norm based on a finite element space. See braid_SetTemporalNorm for information on how the spatial norm is combined over time for a global space-time residual norm. This global norm then controls halting.

10.3.2.15 braid_PtFcnSRefine

```
typedef braid_Int(* braid_PtFcnSRefine) (braid_App app, braid_Vector cu, braid_Vector *fu_ptr, braid_CoarsenRefStatus status)
```

Spatial refinement (optional). Allows the user to refine when going from a coarse time grid to a fine time grid. This function is called on every vector at each level, thus you can refine the entire space time domain. The action of this function should match the braid_PtFcnSCoarsen function.

The user should query the status structure at run time with *braid_CoarsenRefGet**()* calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

10.3.2.16 braid_PtFcnStep

```
typedef braid_Int(* braid_PtFcnStep) (braid_App app, braid_Vector ustop, braid_Vector fstop, braid↩
_Vector u, braid_StepStatus status)
```

Defines the central time stepping function that the user must write.

The user must advance the vector *u* from time *tstart* to *tstop*. The time step is taken assuming the right-hand-side vector *fstop* at time *tstop*. The vector *ustop* may be the same vector as *u* (in the case where not all unknowns are stored). The vector *fstop* is set to NULL to indicate a zero right-hand-side.

Query the status structure with *braid_StepStatusGetTstart(status, &tstart)* and *braid_StepStatusGetTstop(status, &tstop)* to get *tstart* and *tstop*. The status structure also allows for steering. For example, *braid_StepStatusSetR↩ Factor(...)* allows for setting a refinement factor, which tells XBraid to refine this time interval.

10.3.2.17 braid_PtFcnSum

```
typedef braid_Int(* braid_PtFcnSum) (braid_App app, braid_Real alpha, braid_Vector x, braid_Real
beta, braid_Vector y)
```

AXPY, *alpha x* + *beta y* –> *y*

10.3.2.18 braid_PtFcnSync

```
typedef braid_Int(* braid_PtFcnSync) (braid_App app, braid_SyncStatus status)
```

Gives user access to XBraid and to the user's app at various points (primarily once per iteration inside FRefine and outside in the main cycle loop). This function is called once per-processor (not for every state vector stored on the processor, like access).

10.3.2.19 braid_PtFcnTimeGrid

```
typedef braid_Int(* braid_PtFcnTimeGrid) (braid_App app, braid_Real *ta, braid_Int *ilower, braid↩
_Int *iupper)
```

Set time values for temporal grid on level 0 (time slice per processor)

10.3.2.20 braid_Vector

```
typedef struct _braid_Vector_struct* braid_Vector
```

This defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information.

10.4 User-written routines for XBraid_Adjoint

Typedefs

- typedef braid_Int(∗ braid_PtFcnObjectiveT) (braid_App app, braid_Vector u, braid_ObjectiveStatus ostatus, braid_Real ∗objectiveT_ptr)
- typedef braid_Int(∗ braid_PtFcnObjectiveTDiff) (braid_App app, braid_Vector u, braid_Vector u_bar, braid_Real F_bar, braid_ObjectiveStatus ostatus)
- typedef braid_Int(∗ braid_PtFcnPostprocessObjective) (braid_App app, braid_Real sum_obj, braid_Real ∗postprocess_ptr)
- typedef braid_Int(∗ braid_PtFcnPostprocessObjective_diff) (braid_App app, braid_Real sum_obj, braid_Real ∗F_bar_ptr)
- typedef braid_Int(∗ braid_PtFcnStepDiff) (braid_App app, braid_Vector ustop, braid_Vector u, braid_Vector ustop_bar, braid_Vector u_bar, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnResetGradient) (braid_App app)

10.4.1 Detailed Description

These are all the user-written routines needed to use XBraid_Adjoint. There are no new user-written data structures here. But, the braid_App structure will typically be used to store some things like optimization parameters and gradients.

10.4.2 Typedef Documentation

10.4.2.1 braid_PtFcnObjectiveT

```
typedef braid_Int(* braid_PtFcnObjectiveT) (braid_App app, braid_Vector u, braid_ObjectiveStatus
ostatus, braid_Real *objectiveT_ptr)
```

This routine evaluates the time-dependent part of the objective function, at a current time *t*, i.e. the integrand. Query the braid_ObjectiveStatus structure for information about the current time and status of XBraid_Adjoint.

10.4.2.2 braid_PtFcnObjectiveTDiff

```
typedef braid_Int(* braid_PtFcnObjectiveTDiff) (braid_App app, braid_Vector u, braid_Vector u_bar,
braid_Real F_bar, braid_ObjectiveStatus ostatus)
```

This is the differentiated version of the braid_PtFcnObjectiveT routine. It provides the derivatives of ObjectiveT() multiplied by the scalar input *F_bar*.

First output: the derivative with respect to the state vector must be returned to XBraid_Adjoint in *u_bar*.

Second output: The derivative with respect to the design must update the gradient, which is stored in the braid_App.

10.4.2.3 braid_PtFcnPostprocessObjective

```
typedef braid_Int(* braid_PtFcnPostprocessObjective) (braid_App app, braid_Real sum_obj, braid_Real *postprocess_ptr)
```

(Optional) This function can be used to postprocess the time-integral objective function. For example, when inverse design problems are considered, you can use a tracking-type objective function by substracting a target value from *postprocess_ptr*, and squaring the result. Relaxation or penalty terms can also be added to *postprocess_ptr*. For a description of the postprocessing routine, see the Section Objective function evaluation .

10.4.2.4 braid_PtFcnPostprocessObjective_diff

```
typedef braid_Int(* braid_PtFcnPostprocessObjective_diff) (braid_App app, braid_Real sum_obj,
braid_Real *F_bar_ptr)
```

(Optional) Differentiated version of the Postprocessing routine.

First output: Return the partial derivative of the braid_PtFcnPostprocessObjective routine with respect to the time-integral objective function, and placing the result in the scalar value *F_bar_ptr*

Second output: Update the gradient with the partial derivative with respect to the design. Gradients are usually stored in braid_App .

For a description of the postprocessing routine, see the Section Objective function evaluation .

10.4.2.5 braid_PtFcnResetGradient

```
typedef braid_Int(* braid_PtFcnResetGradient) (braid_App app)
```

Set the gradient to zero, which is usually stored in braid_App .

10.4.2.6 braid_PtFcnStepDiff

```
typedef braid_Int(* braid_PtFcnStepDiff) (braid_App app, braid_Vector ustop, braid_Vector u, braid↩
_Vector ustop_bar, braid_Vector u_bar, braid_StepStatus status)
```

This is the differentiated version of the time-stepping routine. It provides the transposed derivatives of *Step()* multiplied by the adjoint input vector *u_bar* (or *ustop_bar*).

First output: the derivative with respect to the state *u* updates the adjoint vector *u_bar* (or *ustop_bar*).

Second output: The derivative with respect to the design must update the gradient, which is stored in braid_App .

10.5 User interface routines

Modules

- General Interface routines
- Interface routines for XBraid_Adjoint
- XBraid status structures
- XBraid status routines
- Inherited XBraid status routines
- XBraid status macros

10.5.1 Detailed Description

These are all the user interface routines.

10.6 General Interface routines

Macros

- #define braid_RAND_MAX 32768

Typedefs

- typedef struct _braid_Core_struct * braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnStep step, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnAccess access, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core *core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_WriteConvHistory (braid_Core core, const char *filename)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetIncrMaxLevels (braid_Core core)
- braid_Int braid_SetSkip (braid_Core core, braid_Int skip)
- braid_Int braid_SetRefine (braid_Core core, braid_Int refine)
- braid_Int braid_SetMaxRefinements (braid_Core core, braid_Int max_refinements)
- braid_Int braid_SetTPointsCutoff (braid_Core core, braid_Int tpoints_cutoff)
- braid_Int braid_SetMinCoarse (braid_Core core, braid_Int min_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCRelaxWt (braid_Core core, braid_Int level, braid_Real Cwt)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetNFMG (braid_Core core, braid_Int k)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)
- braid_Int braid_SetStorage (braid_Core core, braid_Int storage)
- braid_Int braid_SetTemporalNorm (braid_Core core, braid_Int tnorm)
- braid_Int braid_SetResidual (braid_Core core, braid_PtFcnResidual residual)
- braid_Int braid_SetFullRNormRes (braid_Core core, braid_PtFcnResidual residual)
- braid_Int braid_SetTimeGrid (braid_Core core, braid_PtFcnTimeGrid tgrid)
- braid_Int braid_SetPeriodic (braid_Core core, braid_Int periodic)
- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnSCoarsen scoarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnSRefine srefine)
- braid_Int braid_SetSync (braid_Core core, braid_PtFcnSync sync)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)
- braid_Int braid_SetFileIOLevel (braid_Core core, braid_Int io_level)
- braid_Int braid_SetPrintFile (braid_Core core, const char *printfile_name)
- braid_Int braid_SetDefaultPrintFile (braid_Core core)
- braid_Int braid_SetAccessLevel (braid_Core core, braid_Int access_level)

- braid_Int braid_SplitCommworld (const MPI_Comm ∗comm_world, braid_Int px, MPI_Comm ∗comm_x, MPI_Comm ∗comm_t)
- braid_Int braid_SetShell (braid_Core core, braid_PtFcnSInit sinit, braid_PtFcnSClone sclone, braid_PtFcnSFree sfree)
- braid_Int braid_GetNumIter (braid_Core core, braid_Int ∗niter_ptr)
- braid_Int braid_GetRNorms (braid_Core core, braid_Int ∗nrequest_ptr, braid_Real ∗rnorms)
- braid_Int braid_GetNLevels (braid_Core core, braid_Int ∗nlevels_ptr)
- braid_Int braid_GetSpatialAccuracy (braid_StepStatus status, braid_Real loose_tol, braid_Real tight_tol, braid_Real ∗tol_ptr)
- braid_Int braid_SetSeqSoln (braid_Core core, braid_Int seq_soln)
- braid_Int braid_GetMyID (braid_Core core, braid_Int ∗myid_ptr)
- braid_Int braid_Rand (void)

10.6.1 Detailed Description

These are general interface routines, e.g., routines to initialize and run a XBraid solver, or to split a communicator into spatial and temporal components.

10.6.2 Macro Definition Documentation

10.6.2.1 braid_RAND_MAX

```
#define braid_RAND_MAX 32768
```

Machine independent pseudo-random number generator is defined in Braid.c

10.6.3 Typedef Documentation

10.6.3.1 braid_Core

```
typedef struct _braid_Core_struct* braid_Core
```

points to the core structure defined in _braid.h

10.6.4 Function Documentation

10.6.4.1 braid_Destroy()

```
braid_Int braid_Destroy (
            braid_Core core )
```

Clean up and destroy core.

Parameters

core	braid_Core (_braid_Core) struct

10.6.4.2 braid_Drive()

```
braid_Int braid_Drive (
            braid_Core core )
```

Carry out a simulation with XBraid. Integrate in time.

Parameters

core	braid_Core (_braid_Core) struct

10.6.4.3 braid_GetMyID()

```
braid_Int braid_GetMyID (
            braid_Core core,
            braid_Int * myid_ptr )
```

Get the processor's rank.

Parameters

core	braid_Core (_braid_Core) struct
myid_ptr	output: rank of the processor.

10.6.4.4 braid_GetNLevels()

```
braid_Int braid_GetNLevels (
            braid_Core core,
            braid_Int * nlevels_ptr )
```

After Drive() finishes, this returns the number of XBraid levels

Parameters

core	braid_Core (_braid_Core) struct
nlevels_ptr	output, holds the number of XBraid levels

10.6.4.5 braid_GetNumIter()

```
braid_Int braid_GetNumIter (
            braid_Core core,
            braid_Int * niter_ptr )
```

After Drive() finishes, this returns the number of iterations taken.

Parameters

core	braid_Core (_braid_Core) struct
niter_ptr	output, holds number of iterations taken

10.6.4.6 braid_GetRNorms()

```
braid_Int braid_GetRNorms (
            braid_Core core,
            braid_Int * nrequest_ptr,
            braid_Real * rnorms )
```

After Drive() finishes, this returns XBraid residual history. If *nrequest_ptr* is negative, return the last *nrequest_ptr* residual norms. If positive, return the first *nrequest_ptr* residual norms. Upon exit, *nrequest_ptr* holds the number of residuals actually returned.

Parameters

core	braid_Core (_braid_Core) struct
nrequest_ptr	input/output, input: num requested resid norms, output: num actually returned
rnorms	output, holds residual norm history array

10.6.4.7 braid_GetSpatialAccuracy()

```
braid_Int braid_GetSpatialAccuracy (
            braid_StepStatus status,
            braid_Real loose_tol,
            braid_Real tight_tol,
            braid_Real * tol_ptr )
```

Example function to compute a tapered stopping tolerance for implicit time stepping routines, i.e., a tolerance *tol_ptr* for the spatial solves. This tapering only occurs on the fine grid.

This rule must be followed. The same tolerance must be returned over all processors, for a given XBraid and XBraid level. Different levels may have different tolerances and the same level may vary its tolerance from iteration to iteration, but for the same iteration and level, the tolerance must be constant.

This additional rule must be followed. The fine grid tolerance is never reduced (this is important for convergence)

On the fine level,the spatial stopping tolerance *tol_ptr* is interpolated from *loose_tol* to *tight_tol* based on the relationship between *rnorm* / *rnorm0* and *tol*. Remember when *rnorm* / *rnorm0* $<$ *tol*, XBraid halts. Thus, this function lets us have a loose stopping tolerance while the Braid residual is still relatively large, and then we transition to a tight stopping tolerance as the Braid residual is reduced.

If the user has not defined a residual function, tight_tol is always returned.

The loose_tol is always used on coarse grids, excepting the above mentioned residual computations.

This function will normally be called from the user's step routine.

This function is also meant as a guide for users to develop their own routine.

Parameters

status	Current XBraid step status
loose_tol	Loosest allowed spatial solve stopping tol on fine grid
tight_tol	Tightest allowed spatial solve stopping tol on fine grid
tol_ptr	output, holds the computed spatial solve stopping tol

10.6.4.8 braid_Init()

```
braid_Int braid_Init (
            MPI_Comm comm_world,
            MPI_Comm comm,
            braid_Real tstart,
            braid_Real tstop,
            braid_Int ntime,
            braid_App app,
            braid_PtFcnStep step,
            braid_PtFcnInit init,
            braid_PtFcnClone clone,
            braid_PtFcnFree free,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm,
            braid_PtFcnAccess access,
            braid_PtFcnBufSize bufsize,
            braid_PtFcnBufPack bufpack,
            braid_PtFcnBufUnpack bufunpack,
            braid_Core * core_ptr )
```

Create a core object with the required initial data.

This core is used by XBraid for internal data structures. The output is *core_ptr* which points to the newly created braid_Core structure.

Parameters

comm_world	Global communicator for space and time
comm	Communicator for temporal dimension
tstart	start time
tstop	End time
ntime	Initial number of temporal grid values
app	User-defined _braid_App structure
step	User time stepping routine to advance a braid_Vector forward one step
init	Initialize a braid_Vector on the finest temporal grid
clone	Clone a braid_Vector
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
access	Allows access to XBraid and current braid_Vector
bufsize	Computes size for MPI buffer for one braid_Vector
bufpack	Packs MPI buffer to contain one braid_Vector

Parameters

bufunpack	Unpacks MPI buffer into a braid_Vector
core_ptr	Pointer to braid_Core (_braid_Core) struct

10.6.4.9 braid_PrintStats()

```
braid_Int braid_PrintStats (
            braid_Core core )
```

Print statistics after a XBraid run.

Parameters

core	braid_Core (_braid_Core) struct

10.6.4.10 braid_Rand()

```
braid_Int braid_Rand (
            void )
```

Define a machine independent random number generator

10.6.4.11 braid_SetAbsTol()

```
braid_Int braid_SetAbsTol (
            braid_Core core,
            braid_Real atol )
```

Set absolute stopping tolerance.

Recommended option over relative tolerance

Parameters

core	braid_Core (_braid_Core) struct
atol	absolute stopping tolerance

10.6.4.12 braid_SetAccessLevel()

```
braid_Int braid_SetAccessLevel (
            braid_Core core,
            braid_Int access_level )
```

Set access level for XBraid. This controls how often the user's access routine is called.

- Level 0: Never call the user's access routine
- Level 1: Only call the user's access routine after XBraid is finished

- Level 2: Call the user's access routine every iteration and on every level. This is during _braid_FRestrict, during the down-cycle part of a XBraid iteration.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
access_level	desired access_level

10.6.4.13 braid_SetCFactor()

```
braid_Int braid_SetCFactor (
              braid_Core core,
              braid_Int level,
              braid_Int cfactor )
```

Set the coarsening factor *cfactor* on grid *level* (level 0 is the finest grid). The default factor is 2 on all levels. To change the default factor, use *level = -1*.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set coarsening factor on
cfactor	desired coarsening factor

10.6.4.14 braid_SetCRelaxWt()

```
braid_Int braid_SetCRelaxWt (
              braid_Core core,
              braid_Int level,
              braid_Real Cwt )
```

Set the C-relaxation weight on grid *level* (level 0 is the finest grid). The default is 1.0 on all levels. To change the default factor, use *level* $*$ = *-1*.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set *Cwt* on
Cwt	C-relaxation weight to use on *level*

10.6.4.15 braid_SetDefaultPrintFile()

```
braid_Int braid_SetDefaultPrintFile (
              braid_Core core )
```

Use default filename, *braid_runtime.out* for runtime print messages. This function is particularly useful for Fortran codes, where passing filename strings between C and Fortran is troublesome. Level of printing is controlled by braid_SetPrintLevel.

Parameters

core	braid_Core (_braid_Core) struct

10.6.4.16 braid_SetFileIOLevel()

```
braid_Int braid_SetFileIOLevel (
            braid_Core core,
            braid_Int io_level )
```

Set output level for XBraid. This controls how much information is saved to files .

- Level 0: no output
- Level 1: save the cycle in braid.out.cycle

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
io_level	desired output-to-file level

10.6.4.17 braid_SetFMG()

```
braid_Int braid_SetFMG (
            braid_Core core )
```

Once called, XBraid will use FMG (i.e., F-cycles).

Parameters

core	braid_Core (_braid_Core) struct

10.6.4.18 braid_SetFullRNormRes()

```
braid_Int braid_SetFullRNormRes (
            braid_Core core,
            braid_PtFcnResidual residual )
```

Set user-defined residual routine for computing full residual norm (all C/F points).

Parameters

Parameters

core	braid_Core (_braid_Core) struct
residual	function pointer to residual routine

10.6.4.19 braid_SetIncrMaxLevels()

```
braid_Int braid_SetIncrMaxLevels (
            braid_Core core )
```

Increase the max number of multigrid levels after performing a refinement.

10.6.4.20 braid_SetMaxIter()

```
braid_Int braid_SetMaxIter (
            braid_Core core,
            braid_Int max_iter )
```

Set max number of multigrid iterations.

Parameters

core	braid_Core (_braid_Core) struct
max_iter	maximum iterations to allow

10.6.4.21 braid_SetMaxLevels()

```
braid_Int braid_SetMaxLevels (
            braid_Core core,
            braid_Int max_levels )
```

Set max number of multigrid levels.

Parameters

core	braid_Core (_braid_Core) struct
max_levels	maximum levels to allow

10.6.4.22 braid_SetMaxRefinements()

```
braid_Int braid_SetMaxRefinements (
            braid_Core core,
            braid_Int max_refinements )
```

Set the max number of time grid refinement levels allowed.

Parameters

core	braid_Core (_braid_Core) struct
max_refinements	maximum refinement levels allowed

10.6.4.23 braid_SetMinCoarse()

```
braid_Int braid_SetMinCoarse (
            braid_Core core,
            braid_Int min_coarse )
```

Set minimum allowed coarse grid size. XBraid stops coarsening whenever creating the next coarser grid will result in a grid smaller than min_coarse. The maximum possible coarse grid size will be min_coarse∗coarsening_factor.

Parameters

core	braid_Core (_braid_Core) struct
min_coarse	minimum coarse grid size

10.6.4.24 braid_SetNFMG()

```
braid_Int braid_SetNFMG (
            braid_Core core,
            braid_Int k )
```

Once called, XBraid will use FMG (i.e., F-cycles.

Parameters

core	braid_Core (_braid_Core) struct
k	number of initial F-cycles to do before switching to V-cycles

10.6.4.25 braid_SetNFMGVcyc()

```
braid_Int braid_SetNFMGVcyc (
            braid_Core core,
            braid_Int nfmg_Vcyc )
```

Set number of V-cycles to use at each FMG level (standard is 1)

Parameters

core	braid_Core (_braid_Core) struct
nfmg_Vcyc	number of V-cycles to do each FMG level

10.6.4.26 braid_SetNRelax()

```
braid_Int braid_SetNRelax (
            braid_Core core,
            braid_Int level,
            braid_Int nrelax )
```

Set the number of relaxation sweeps *nrelax* on grid *level* (level 0 is the finest grid). The default is 1 on all levels. To change the default factor, use *level* = *-1*. One sweep is a CF relaxation sweep.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set *nrelax* on
nrelax	number of relaxations to do on *level*

10.6.4.27 braid_SetPeriodic()

```
braid_Int braid_SetPeriodic (
            braid_Core core,
            braid_Int periodic )
```

Set periodic time grid. The periodicity on each grid level is given by the number of points on each level. Requirements: The number of points on the finest grid level must be evenly divisible by the product of the coarsening factors between each grid level. Currently, the coarsening factors must be the same on all grid levels. Also, braid_SetSeqSoln must not be used.

Parameters

core	braid_Core (_braid_Core) struct
periodic	boolean to specify if periodic

10.6.4.28 braid_SetPrintFile()

```
braid_Int braid_SetPrintFile (
            braid_Core core,
            const char * printfile_name )
```

Set output file for runtime print messages. Level of printing is controlled by braid_SetPrintLevel. Default is stdout.

Parameters

core	braid_Core (_braid_Core) struct
printfile_name	output file for XBraid runtime output

10.6.4.29 braid_SetPrintLevel()

```
braid_Int braid_SetPrintLevel (
```

```
braid_Core core,
braid_Int print_level )
```

Set print level for XBraid. This controls how much information is printed to the XBraid print file (braid_SetPrintFile).

- Level 0: no output
- Level 1: print runtime information like the residual history
- Level 2: level 1 output, plus post-Braid run statistics (default)
- Level 3: level 2 output, plus debug level output.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
print_level	desired print level

10.6.4.30 braid_SetRefine()

```
braid_Int braid_SetRefine (
            braid_Core core,
            braid_Int refine )
```

Turn time refinement on (refine = 1) or off (refine = 0).

Parameters

core	braid_Core (_braid_Core) struct
refine	boolean, refine in time or not

10.6.4.31 braid_SetRelTol()

```
braid_Int braid_SetRelTol (
            braid_Core core,
            braid_Real rtol )
```

Set relative stopping tolerance, relative to the initial residual. Be careful. If your initial guess is all zero, then the initial residual may only be nonzero over one or two time values, and this will skew the relative tolerance. Absolute tolerances are recommended.

Parameters

core	braid_Core (_braid_Core) struct
rtol	relative stopping tolerance

10.6.4.32 braid_SetResidual()

```
braid_Int braid_SetResidual (
            braid_Core core,
            braid_PtFcnResidual residual )
```

Set user-defined residual routine.

Parameters

core	braid_Core (_braid_Core) struct
residual	function pointer to residual routine

10.6.4.33 braid_SetSeqSoln()

```
braid_Int braid_SetSeqSoln (
            braid_Core core,
            braid_Int seq_soln )
```

Set the initial guess to XBraid as the sequential time stepping solution. This is primarily for debugging. When used with storage=-2, the initial residual should evaluate to exactly 0. The residual can also be 0 for other storage options if the time stepping is *exact*, e.g., the implicit solve in Step is done to full precision.

The value *seq_soln* is a Boolean

- 0: The user's Init() function initializes the state vector (default)
- 1: Sequential time stepping, with the user's initial condition from Init(t=0) initializes the state vector

Default is 0.

Parameters

core	braid_Core (_braid_Core) struct
seq_soln	1: Init with sequential time stepping soln, 0: Use user's Init()

10.6.4.34 braid_SetShell()

```
braid_Int braid_SetShell (
            braid_Core core,
            braid_PtFcnSInit sinit,
            braid_PtFcnSClone sclone,
            braid_PtFcnSFree sfree )
```

Activate the shell vector feature, and set the various functions that are required :

- sinit : create a shell vector
- sclone : clone the shell of a vector
- sfree : free the data of a vector, keeping its shell This feature should be used with storage option = -1. It allows the used to keep metadata on all points (including F-points) without storing the all vector everywhere. With these options, the vectors are fully stored on C-points, but only the vector shell is kept on F-points.

10.6.4.35 braid_SetSkip()

```
braid_Int braid_SetSkip (
            braid_Core core,
            braid_Int skip )
```

Set whether to skip all work on the first down cycle (skip = 1). On by default.

Parameters

core	braid_Core (_braid_Core) struct
skip	boolean, whether to skip all work on first down-cycle

10.6.4.36 braid_SetSpatialCoarsen()

```
braid_Int braid_SetSpatialCoarsen (
            braid_Core core,
            braid_PtFcnSCoarsen scoarsen )
```

Set spatial coarsening routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
scoarsen	function pointer to spatial coarsening routine

10.6.4.37 braid_SetSpatialRefine()

```
braid_Int braid_SetSpatialRefine (
            braid_Core core,
            braid_PtFcnSRefine srefine )
```

Set spatial refinement routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
srefine	function pointer to spatial refinement routine

10.6.4.38 braid_SetStorage()

```
braid_Int braid_SetStorage (
            braid_Core core,
            braid_Int storage )
```

Sets the storage properties of the code. -1 : Default, store only C-points 0 : Full storage of C- and F-Points on all levels x > 0 : Full storage on all levels >= x

Parameters

core	braid_Core (_braid_Core) struct
storage	storage property

10.6.4.39 braid_SetSync()

```
braid_Int braid_SetSync (
            braid_Core core,
            braid_PtFcnSync sync )
```

Set sync routine with user-defined routine. Sync gives user access to XBraid and the user's app at various points (primarily once per iteration inside FRefine and outside in the main cycle loop). This function is called once per-processor (instead of for every state vector on the processor, like access). The use case is to allow the user to update their app once-per iteration based on information from XBraid, for example to maintain the space-time grid when doing time-space adaptivity. Default is no sync routine.

Parameters

core	braid_Core (_braid_Core) struct
sync	function pointer to sync routine

10.6.4.40 braid_SetTemporalNorm()

```
braid_Int braid_SetTemporalNorm (
            braid_Core core,
            braid_Int tnorm )
```

Sets XBraid temporal norm.

This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by braid_PtFcnSpatialNorm at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.

There are three options for setting *tnorm*. See section Halting tolerance for a more detailed discussion (in Introduction.md).

- *tnorm=1*: One-norm summation of spatial norms
- *tnorm=2*: Two-norm summation of spatial norms
- *tnorm=3*: Infinity-norm combination of spatial norms

The default choice is *tnorm=2*

Parameters

core	braid_Core (_braid_Core) struct
tnorm	choice of temporal norm

10.6.4.41 braid_SetTimeGrid()

```
braid_Int braid_SetTimeGrid (
            braid_Core core,
            braid_PtFcnTimeGrid tgrid )
```

Set user-defined time points on finest grid

Parameters

core	braid_Core (_braid_Core) struct
tgrid	function pointer to time grid routine

10.6.4.42 braid_SetTPointsCutoff()

```
braid_Int braid_SetTPointsCutoff (
            braid_Core core,
            braid_Int tpoints_cutoff )
```

Set the number of time steps, beyond which refinements stop. If num(tpoints) $>$ tpoints_cutoff, then stop doing refinements.

Parameters

core	braid_Core (_braid_Core) struct
tpoints_cutoff	cutoff for stopping refinements

10.6.4.43 braid_SplitCommworld()

```
braid_Int braid_SplitCommworld (
            const MPI_Comm * comm_world,
            braid_Int px,
            MPI_Comm * comm_x,
            MPI_Comm * comm_t )
```

Split MPI commworld into *comm_x* and *comm_t*, the spatial and temporal communicators. The total number of processors will equal Px$*$Pt, there Px is the number of procs in space, and Pt is the number of procs in time.

Parameters

comm_world	Global communicator to split
px	Number of processors parallelizing space for a single time step
comm_x	Spatial communicator (written as output)
comm_t	Temporal communicator (written as output)

10.6.4.44 braid_WriteConvHistory()

```
braid_Int braid_WriteConvHistory (
            braid_Core core,
            const char * filename )
```

After Drive() finishes, this function can be called to write out the convergence history (residuals for each iteration) to a file

Parameters

core	braid_Core (_braid_Core) struct
filename	Output file name

10.7 Interface routines for XBraid_Adjoint

Functions

- braid_Int braid_InitAdjoint (braid_PtFcnObjectiveT objectiveT, braid_PtFcnObjectiveTDiff objectiveT_diff, braid_PtFcnStepDiff step_diff, braid_PtFcnResetGradient reset_gradient, braid_Core *core_ptr)
- braid_Int braid_SetTStartObjective (braid_Core core, braid_Real tstart_obj)
- braid_Int braid_SetTStopObjective (braid_Core core, braid_Real tstop_obj)
- braid_Int braid_SetPostprocessObjective (braid_Core core, braid_PtFcnPostprocessObjective post_fcn)
- braid_Int braid_SetPostprocessObjective_diff (braid_Core core, braid_PtFcnPostprocessObjective_diff post_fcn_diff)
- braid_Int braid_SetAbsTolAdjoint (braid_Core core, braid_Real tol_adj)
- braid_Int braid_SetRelTolAdjoint (braid_Core core, braid_Real rtol_adj)
- braid_Int braid_SetObjectiveOnly (braid_Core core, braid_Int boolean)
- braid_Int braid_GetObjective (braid_Core core, braid_Real *objective_ptr)
- braid_Int braid_GetRNormAdjoint (braid_Core core, braid_Real *rnorm_adj)
- braid_Int braid_SetRichardsonEstimation (braid_Core core, braid_Int est_error, braid_Int richardson, braid_Int local_order)

10.7.1 Detailed Description

These are interface routines for computing adjoint sensitivities, i.e., adjoint-based gradients. These routines initialize the XBraid_Adjoint solver, and allow the user to set XBraid_Adjoint solver parameters.

10.7.2 Function Documentation

10.7.2.1 braid_GetObjective()

```
braid_Int braid_GetObjective (
            braid_Core core,
            braid_Real * objective_ptr )
```

After braid_Drive has finished, this returns the objective function value.

Parameters

core	braid_Core struct
objective_ptr	output: value of the objective function

10.7.2.2 braid_GetRNormAdjoint()

```
braid_Int braid_GetRNormAdjoint (
            braid_Core core,
            braid_Real * rnorm_adj )
```

After braid_Drive has finished, this returns the residual norm after the last XBraid iteration.

Parameters

core	braid_Core struct
rnorm_adj	output: adjoint residual norm of last iteration

10.7.2.3 braid_InitAdjoint()

```
braid_Int braid_InitAdjoint (
            braid_PtFcnObjectiveT objectiveT,
            braid_PtFcnObjectiveTDiff objectiveT_diff,
            braid_PtFcnStepDiff step_diff,
            braid_PtFcnResetGradient reset_gradient,
            braid_Core * core_ptr )
```

Initialize the XBraid_Adjoint solver for computing adjoint sensitivities. Once this function is called, braid_Drive will then compute gradient information alongside the primal XBraid computations.

Parameters

objectiveT	user-routine: evaluates the time-dependent objective function value at time *t*
objectiveT_diff	user-routine: differentiated version of the objectiveT function
step_diff	user-routine: differentiated version of the step function
reset_gradient	user-routine: set the gradient to zero (storage location of gradient up to user)
core_ptr	pointer to braid_Core (_braid_Core) struct

10.7.2.4 braid_SetAbsTolAdjoint()

```
braid_Int braid_SetAbsTolAdjoint (
            braid_Core core,
            braid_Real tol_adj )
```

Set an absolute halting tolerance for the adjoint residuals. XBraid_Adjoint stops iterating when the adjoint residual is below this value.

Parameters

core	braid_Core (_braid_Core) struct
tol_adj	absolute stopping tolerance for adjoint solve

10.7.2.5 braid_SetObjectiveOnly()

```
braid_Int braid_SetObjectiveOnly (
            braid_Core core,
            braid_Int boolean )
```

Set this option with *boolean = 1*, and then *braid_Drive(core)* will skip the gradient computation and only compute the forward ODE solution and objective function value. Reset this option with *boolean = 0* to turn the adjoint solve and gradient computations back on.

Parameters

core	braid_Core (_braid_Core) struct
boolean	set to '1' for computing objective function only, '0' for computing objective function AND gradients

10.7.2.6 braid_SetPostprocessObjective()

```
braid_Int braid_SetPostprocessObjective (
            braid_Core core,
            braid_PtFcnPostprocessObjective post_fcn )
```

Pass the postprocessing objective function *F* to XBraid_Adjoint. For a description of *F*, see the Section Objective function evaluation .

Parameters

core	braid_Core (_braid_Core) struct
post_fcn	function pointer to postprocessing routine

10.7.2.7 braid_SetPostprocessObjective_diff()

```
braid_Int braid_SetPostprocessObjective_diff (
            braid_Core core,
            braid_PtFcnPostprocessObjective_diff post_fcn_diff )
```

Pass the differentiated version of the postprocessing objective function *F* to XBraid_Adjoint. For a description of *F*, see the Section Objective function evaluation .

Parameters

core	braid_Core (_braid_Core) struct
post_fcn_diff	function pointer to differentiated postprocessing routine

10.7.2.8 braid_SetRelTolAdjoint()

```
braid_Int braid_SetRelTolAdjoint (
            braid_Core core,
            braid_Real rtol_adj )
```

Set a relative stopping tolerance for adjoint residuals. XBraid_Adjoint will stop iterating when the relative residual drops below this value. Be careful when using a relative stopping criterion. The initial residual may already be close to zero, and this will skew the relative tolerance. Absolute tolerances are recommended.

Parameters

core	braid_Core (_braid_Core) struct

Parameters

rtol_adj	relative stopping tolerance for adjoint solve

10.7.2.9 braid_SetRichardsonEstimation()

```
braid_Int braid_SetRichardsonEstimation (
            braid_Core core,
            braid_Int est_error,
            braid_Int richardson,
            braid_Int local_order )
```

Turn on built-in Richardson-based error estimation and/or extrapolation with XBraid. When enabled, the Richardson extrapolation (RE) option (richardson == 1) is used to improve the accuracy of the solution at the C-points on the finest level. When the built-in error estimate option is turned on (est_error == 1), RE is used to estimate the local truncation error at each point. These estimates can be accessed through StepStatus and AccessStatus functions.

The last parameter is local_order, which represents the LOCAL order of the time integration scheme. e.g. local_order = 2 for Backward Euler.

Also, the Richardson error estimate is only available after roughly 1 Braid iteration. The estimate is given a dummy value of -1.0, until an actual estimate is available. Thus after an adaptive refinement, and a new hierarchy is formed, another iteration must pass before the error estimates are available again.

Parameters

core	braid_Core (_braid_Core) struct
est_error	Boolean, if 1 compute Richardson-based error estimates, if 0, then do not
richardson	Boolean, if 1 carry out Richardson-based extrapolation to enhance accuracy on the fine-grid, if 0, then do not
local_order	Local order of the time integration scheme, e.g., local _order=2 for backward Euler

10.7.2.10 braid_SetTStartObjective()

```
braid_Int braid_SetTStartObjective (
            braid_Core core,
            braid_Real tstart_obj )
```

Set a start time for integrating the objective function over time. Default is *tstart* of the primal XBraid run.

Parameters

core	braid_Core (_braid_Core) struct
tstart_obj	time value for starting the time-integration of the objective function

10.7.2.11 braid_SetTStopObjective()

```
braid_Int braid_SetTStopObjective (
```

```
braid_Core core,
braid_Real tstop_obj )
```

Set the end-time for integrating the objective function over time. Default is *tstop* of the primal XBraid run

Parameters

core	braid_Core (_braid_Core) struct
tstop_obj	time value for stopping the time-integration of the objective function

10.8 XBraid status structures

Typedefs

- typedef struct _braid_Status_struct * braid_Status
- typedef struct _braid_AccessStatus_struct * braid_AccessStatus
- typedef struct _braid_SyncStatus_struct * braid_SyncStatus
- typedef struct _braid_StepStatus_struct * braid_StepStatus
- typedef struct _braid_CoarsenRefStatus_struct * braid_CoarsenRefStatus
- typedef struct _braid_BufferStatus_struct * braid_BufferStatus
- typedef struct _braid_ObjectiveStatus_struct * braid_ObjectiveStatus

10.8.1 Detailed Description

Define the different status types.

10.8.2 Typedef Documentation

10.8.2.1 braid_AccessStatus

```
typedef struct _braid_AccessStatus_struct* braid_AccessStatus
```

AccessStatus structure which defines the status of XBraid at a given instant on some level during a run. The user accesses it through *braid_AccessStatusGet**()* functions. This is just a pointer to the braid_Status.

10.8.2.2 braid_BufferStatus

```
typedef struct _braid_BufferStatus_struct* braid_BufferStatus
```

The user's bufpack, bufunpack and bufsize routines will receive a BufferStatus structure, which defines the status of XBraid at a given buff (un)pack instance. The user accesses it through *braid_BufferStatusGet**()* functions. This is just a pointer to the braid_Status.

10.8.2.3 braid_CoarsenRefStatus

```
typedef struct _braid_CoarsenRefStatus_struct* braid_CoarsenRefStatus
```

The user coarsen and refine routines will receive a CoarsenRefStatus structure, which defines the status of XBraid at a given instant of coarsening or refinement on some level during a run. The user accesses it through *braid_CoarsenRefStatusGet**()* functions. This is just a pointer to the braid_Status.

10.8.2.4 braid_ObjectiveStatus

```
typedef struct _braid_ObjectiveStatus_struct* braid_ObjectiveStatus
```

The user's objectiveT and PostprocessObjective will receive an ObjectiveStatus structure, which defines the status of XBraid at a given instance of evaluating the objective function. The user accesses it through *braid_ObjectiveStatusGet**()* functions. This is just a pointer to the braid_Status.

10.8.2.5 braid_Status

```
typedef struct _braid_Status_struct* braid_Status
```

This is the main Status structure, that contains the properties of all the status. The user does not have access to this structure, but only to the derived Status structures. This class is accessed only inside XBraid code.

10.8.2.6 braid_StepStatus

```
typedef struct _braid_StepStatus_struct* braid_StepStatus
```

The user's step routine routine will receive a StepStatus structure, which defines the status of XBraid at the given instant for step evaluation on some level during a run. The user accesses it through *braid_StepStatusGet**()* functions. This is just a pointer to the braid_Status.

10.8.2.7 braid_SyncStatus

```
typedef struct _braid_SyncStatus_struct* braid_SyncStatus
```

SyncStatus structure which provides the status of XBraid at a given instant on some level during a run. This is vector independent and called once per processor. The user accesses it through *braid_SyncStatusGet**()* functions. This is just a pointer to the braid_Status.

10.9 XBraid status routines

Functions

- braid_Int braid_StatusGetT (braid_Status status, braid_Real ∗t_ptr)
- braid_Int braid_StatusGetTIndex (braid_Status status, braid_Int ∗idx_ptr)
- braid_Int braid_StatusGetIter (braid_Status status, braid_Int ∗iter_ptr)
- braid_Int braid_StatusGetLevel (braid_Status status, braid_Int ∗level_ptr)
- braid_Int braid_StatusGetNLevels (braid_Status status, braid_Int ∗nlevels_ptr)
- braid_Int braid_StatusGetNRefine (braid_Status status, braid_Int ∗nrefine_ptr)
- braid_Int braid_StatusGetNTPoints (braid_Status status, braid_Int ∗ntpoints_ptr)
- braid_Int braid_StatusGetResidual (braid_Status status, braid_Real ∗rnorm_ptr)
- braid_Int braid_StatusGetDone (braid_Status status, braid_Int ∗done_ptr)
- braid_Int braid_StatusGetTIUL (braid_Status status, braid_Int ∗iloc_upper, braid_Int ∗iloc_lower, braid_Int level)
- braid_Int braid_StatusGetTimeValues (braid_Status status, braid_Real ∗∗tvalues_ptr, braid_Int i_upper, braid_Int i_lower, braid_Int level)
- braid_Int braid_StatusGetTILD (braid_Status status, braid_Real ∗t_ptr, braid_Int ∗iter_ptr, braid_Int ∗level_ptr, braid_Int ∗done_ptr)
- braid_Int braid_StatusGetWrapperTest (braid_Status status, braid_Int ∗wtest_ptr)
- braid_Int braid_StatusGetCallingFunction (braid_Status status, braid_Int ∗cfunction_ptr)
- braid_Int braid_StatusGetCTprior (braid_Status status, braid_Real ∗ctprior_ptr)
- braid_Int braid_StatusGetCTstop (braid_Status status, braid_Real ∗ctstop_ptr)
- braid_Int braid_StatusGetFTprior (braid_Status status, braid_Real ∗ftprior_ptr)
- braid_Int braid_StatusGetFTstop (braid_Status status, braid_Real ∗ftstop_ptr)
- braid_Int braid_StatusGetTpriorTstop (braid_Status status, braid_Real ∗t_ptr, braid_Real ∗ftprior_ptr, braid_Real ∗ftstop_ptr, braid_Real ∗ctprior_ptr, braid_Real ∗ctstop_ptr)
- braid_Int braid_StatusGetTstop (braid_Status status, braid_Real ∗tstop_ptr)
- braid_Int braid_StatusGetTstartTstop (braid_Status status, braid_Real ∗tstart_ptr, braid_Real ∗tstop_ptr)
- braid_Int braid_StatusGetTol (braid_Status status, braid_Real ∗tol_ptr)
- braid_Int braid_StatusGetRNorms (braid_Status status, braid_Int ∗nrequest_ptr, braid_Real ∗rnorms_ptr)
- braid_Int braid_StatusGetOldFineTolx (braid_Status status, braid_Real ∗old_fine_tolx_ptr)
- braid_Int braid_StatusSetOldFineTolx (braid_Status status, braid_Real old_fine_tolx)
- braid_Int braid_StatusSetTightFineTolx (braid_Status status, braid_Real tight_fine_tolx)
- braid_Int braid_StatusSetRFactor (braid_Status status, braid_Real rfactor)
- braid_Int braid_StatusSetRefinementDtValues (braid_Status status, braid_Real rfactor, braid_Real ∗dtarray)
- braid_Int braid_StatusSetRSpace (braid_Status status, braid_Real r_space)
- braid_Int braid_StatusGetMessageType (braid_Status status, braid_Int ∗messagetype_ptr)
- braid_Int braid_StatusSetSize (braid_Status status, braid_Real size)
- braid_Int braid_StatusGetSingleErrorEstStep (braid_Status status, braid_Real ∗estimate)
- braid_Int braid_StatusGetSingleErrorEstAccess (braid_Status status, braid_Real ∗estimate)
- braid_Int braid_StatusGetNumErrorEst (braid_Status status, braid_Int ∗npoints)
- braid_Int braid_StatusGetAllErrorEst (braid_Status status, braid_Real ∗error_est)

10.9.1 Detailed Description

XBraid status structures and associated Get/Set routines are what tell the user the status of the simulation when their routines (step, coarsen/refine, access) are called.

10.9.2 Function Documentation

10.9.2.1 braid_StatusGetAllErrorEst()

```
braid_Int braid_StatusGetAllErrorEst (
            braid_Status status,
            braid_Real * error_est )
```

Get All the Richardson based error estimates, e.g. from inside Sync. Use this function in conjuction with GetNumError←Est(). Workflow: use GetNumErrorEst() to get the size of the needed user-array that will hold the error estimates, then pre-allocate array, then call this function to write error estimates to the user-array, then post-process array in user-code. This post-processing will often occur in the Sync function. See examples/ex-06.c.

The error_est array must be user-allocated.

Parameters

status	structure containing current simulation info
error_est	output, user-allocated error estimate array, written by Braid, equals -1 if not available yet (e.g., before iteration 1, or after refinement)

10.9.2.2 braid_StatusGetCallingFunction()

```
braid_Int braid_StatusGetCallingFunction (
            braid_Status status,
            braid_Int * cfunction_ptr )
```

Return flag indicating from which function the vector is accessed

Parameters

status	structure containing current simulation info
cfunction_ptr	output, function number (0=FInterp, 1=FRestrict, 2=FRefine, 3=FAccess, 4=FRefine after refinement, 5=Drive Top of Cycle)

10.9.2.3 braid_StatusGetCTprior()

```
braid_Int braid_StatusGetCTprior (
            braid_Status status,
            braid_Real * ctprior_ptr )
```

Return the **coarse grid** time value to the left of the current time value from the Status structure.

Parameters

status	structure containing current simulation info
ctprior_ptr	output, time value to the left of current time value on coarse grid

10.9.2.4 braid_StatusGetCTstop()

```
braid_Int braid_StatusGetCTstop (
            braid_Status status,
            braid_Real * ctstop_ptr )
```

Return the **coarse grid** time value to the right of the current time value from the Status structure.

Parameters

status	structure containing current simulation info
ctstop_ptr	output, time value to the right of current time value on coarse grid

10.9.2.5 braid_StatusGetDone()

```
braid_Int braid_StatusGetDone (
            braid_Status status,
            braid_Int * done_ptr )
```

Return whether XBraid is done for the current simulation.

done_ptr = 1 indicates that XBraid has finished iterating, (either maxiter has been reached, or the tolerance has been met).

Parameters

status	structure containing current simulation info
done_ptr	output, =1 if XBraid has finished, else =0

10.9.2.6 braid_StatusGetFTprior()

```
braid_Int braid_StatusGetFTprior (
            braid_Status status,
            braid_Real * ftprior_ptr )
```

Return the **fine grid** time value to the left of the current time value from the Status structure.

Parameters

status	structure containing current simulation info
ftprior_ptr	output, time value to the left of current time value on fine grid

10.9.2.7 braid_StatusGetFTstop()

```
braid_Int braid_StatusGetFTstop (
            braid_Status status,
            braid_Real * ftstop_ptr )
```

Return the **fine grid** time value to the right of the current time value from the Status structure.

Parameters

status	structure containing current simulation info
ftstop_ptr	output, time value to the right of current time value on fine grid

10.9.2.8 braid_StatusGetIter()

```
braid_Int braid_StatusGetIter (
            braid_Status status,
            braid_Int * iter_ptr )
```

Return the current iteration from the Status structure.

Parameters

status	structure containing current simulation info
iter_ptr	output, current XBraid iteration number

10.9.2.9 braid_StatusGetLevel()

```
braid_Int braid_StatusGetLevel (
            braid_Status status,
            braid_Int * level_ptr )
```

Return the current XBraid level from the Status structure.

Parameters

status	structure containing current simulation info
level_ptr	output, current level in XBraid

10.9.2.10 braid_StatusGetMessageType()

```
braid_Int braid_StatusGetMessageType (
            braid_Status status,
            braid_Int * messagetype_ptr )
```

Return the current message type from the Status structure.

Parameters

status	structure containing current simulation info
messagetype_ptr	output, type of message, 0: for Step(), 1: for load balancing

10.9.2.11 braid_StatusGetNLevels()

```
braid_Int braid_StatusGetNLevels (
            braid_Status status,
            braid_Int * nlevels_ptr )
```

Return the total number of XBraid levels from the Status structure.

Parameters

status	structure containing current simulation info
nlevels_ptr	output, number of levels in XBraid

10.9.2.12 braid_StatusGetNRefine()

```
braid_Int braid_StatusGetNRefine (
            braid_Status status,
            braid_Int * nrefine_ptr )
```

Return the number of refinements done.

Parameters

status	structure containing current simulation info
nrefine_ptr	output, number of refinements done

10.9.2.13 braid_StatusGetNTPoints()

```
braid_Int braid_StatusGetNTPoints (
            braid_Status status,
            braid_Int * ntpoints_ptr )
```

Return the global number of time points on the fine grid.

Parameters

status	structure containing current simulation info
ntpoints_ptr	output, number of time points on the fine grid

10.9.2.14 braid_StatusGetNumErrorEst()

```
braid_Int braid_StatusGetNumErrorEst (
            braid_Status status,
            braid_Int * npoints )
```

Get the number of local Richardson-based error estimates stored on this processor. Use this function in conjuction with GetAllErrorEst(). Workflow: use this function to get the size of the needed user-array that will hold the error estimates, then pre-allocate array, then call GetAllErrorEst() to write error estimates to the user-array, then post-process array in user-code. This post-processing will often occur in the Sync function. See examples/ex-06.c.

Parameters

status	structure containing current simulation info
npoints	output, number of locally stored Richardson error estimates

10.9.2.15 braid_StatusGetOldFineTolx()

```
braid_Int braid_StatusGetOldFineTolx (
            braid_Status status,
            braid_Real * old_fine_tolx_ptr )
```

Return the previous *old_fine_tolx* set through *braid_StatusSetOldFineTolx* This is used especially by *braid_Get↩ SpatialAccuracy

Parameters

status	structure containing current simulation info
old_fine_tolx_ptr	output, previous *old_fine_tolx*, set through *braid_StepStatusSetOldFineTolx*

10.9.2.16 braid_StatusGetResidual()

```
braid_Int braid_StatusGetResidual (
            braid_Status status,
            braid_Real * rnorm_ptr )
```

Return the current residual norm from the Status structure.

Parameters

status	structure containing current simulation info
rnorm_ptr	output, current residual norm

10.9.2.17 braid_StatusGetRNorms()

```
braid_Int braid_StatusGetRNorms (
            braid_Status status,
            braid_Int * nrequest_ptr,
            braid_Real * rnorms_ptr )
```

Return the current XBraid residual history. If *nrequest_ptr* is negative, return the last *nrequest_ptr* residual norms. If positive, return the first *nrequest_ptr* residual norms. Upon exit, *nrequest_ptr* holds the number of residuals actually returned.

Parameters

status	structure containing current simulation info

Parameters

nrequest_ptr	input/output, input: number of requested residual norms, output: number actually copied
rnorms_ptr	output, XBraid residual norm history, of length *nrequest_ptr*

10.9.2.18 braid_StatusGetSingleErrorEstAccess()

```
braid_Int braid_StatusGetSingleErrorEstAccess (
            braid_Status status,
            braid_Real * estimate )
```

Get the Richardson based error estimate at the single time point currently accessible from Access.

Note that Access needs specific logic distinct from Step, hence please use braid_StepStatusGetSingleErrorEstStep for the user Step() function.

Parameters

status	structure containing current simulation info
estimate	output, error estimate, equals -1 if not available yet (e.g., before iteration 1, or after refinement)

10.9.2.19 braid_StatusGetSingleErrorEstStep()

```
braid_Int braid_StatusGetSingleErrorEstStep (
            braid_Status status,
            braid_Real * estimate )
```

Get the Richardson based error estimate at the single time point currently being "Stepped", i.e., return the current error estimate for the time point at "tstart".

Note that Step needs specific logic distinct from Access, hence please use braid_AccessStatusGetSingleErrorEstAccess for the user Access() function.

Parameters

status	structure containing current simulation info
estimate	output, error estimate, equals -1 if not available yet (e.g., before iteration 1, or after refinement)

10.9.2.20 braid_StatusGetT()

```
braid_Int braid_StatusGetT (
            braid_Status status,
            braid_Real * t_ptr )
```

Return the current time from the Status structure.

Parameters

status	structure containing current simulation info

Parameters

t_ptr	output, current time

10.9.2.21 braid_StatusGetTILD()

```
braid_Int braid_StatusGetTILD (
            braid_Status status,
            braid_Real * t_ptr,
            braid_Int * iter_ptr,
            braid_Int * level_ptr,
            braid_Int * done_ptr )
```

Return XBraid status for the current simulation. Four values are returned.

TILD : time, iteration, level, done

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_StatusGetDone* for more information on the *done* value.

Parameters

status	structure containing current simulation info
t_ptr	output, current time
iter_ptr	output, current XBraid iteration number
level_ptr	output, current level in XBraid
done_ptr	output, =1 if XBraid has finished, else =0

10.9.2.22 braid_StatusGetTimeValues()

```
braid_Int braid_StatusGetTimeValues (
            braid_Status status,
            braid_Real ** tvalues_ptr,
            braid_Int i_upper,
            braid_Int i_lower,
            braid_Int level )
```

Returns an array of time values corresponding to the given inputs. The inputs are the level you want the time values from, the upper time point index you want the value of, and the lower time point index you want the time value of. The output is then filled with all time values from the upper index to the lower index, inclusive.

The caller is responsible for allocating and managing the memory for the array. Time values are filled in so that tvalues↩_ptr[0] corresponds to the lower time index.

Parameters

status	structure containing current simulation info
tvalues_ptr	output, time point values for the requested range of indices
i_upper	input, upper index of the desired time value range (inclusive)
i_lower	input, lower index of the desired time value range (inclusive)
level	input, level for the desired time values

10.9.2.23 braid_StatusGetTIndex()

```
braid_Int braid_StatusGetTIndex (
            braid_Status status,
            braid_Int * idx_ptr )
```

Return the index value corresponding to the current time value from the Status structure.

For Step(), this corresponds to the time-index of "tstart", as this is the time-index of the input vector. That is, NOT the time-index of "tstop". For Access, this corresponds just simply to the time-index of the input vector.

Parameters

status	structure containing current simulation info
idx_ptr	output, global index value corresponding to current time value

10.9.2.24 braid_StatusGetTIUL()

```
braid_Int braid_StatusGetTIUL (
            braid_Status status,
            braid_Int * iloc_upper,
            braid_Int * iloc_lower,
            braid_Int level )
```

Returns upper and lower time point indices on this processor. Two values are returned. Requires the user to specify which level they want the time point indices from.

Parameters

status	structure containing current simulation info
iloc_upper	output, the upper time point index on this processor
iloc_lower	output, the lower time point index on this processor
level	input, level for the desired indices

10.9.2.25 braid_StatusGetTol()

```
braid_Int braid_StatusGetTol (
            braid_Status status,
            braid_Real * tol_ptr )
```

Return the current XBraid stopping tolerance

Parameters

status	structure containing current simulation info
tol_ptr	output, current XBraid stopping tolerance

10.9.2.26 braid_StatusGetTpriorTstop()

```
braid_Int braid_StatusGetTpriorTstop (
            braid_Status status,
            braid_Real * t_ptr,
            braid_Real * ftprior_ptr,
            braid_Real * ftstop_ptr,
            braid_Real * ctprior_ptr,
            braid_Real * ctstop_ptr )
```

Return XBraid status for the current simulation. Five values are returned, tstart, f_tprior, f_tstop, c_tprior, c_tstop.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_StatusGetCTprior* for more information on the *c_tprior* value.

Parameters

status	structure containing current simulation info
t_ptr	output, current time
ftprior_ptr	output, time value to the left of current time value on fine grid
ftstop_ptr	output, time value to the right of current time value on fine grid
ctprior_ptr	output, time value to the left of current time value on coarse grid
ctstop_ptr	output, time value to the right of current time value on coarse grid

10.9.2.27 braid_StatusGetTstartTstop()

```
braid_Int braid_StatusGetTstartTstop (
            braid_Status status,
            braid_Real * tstart_ptr,
            braid_Real * tstop_ptr )
```

Return XBraid status for the current simulation. Two values are returned, tstart and tstop.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_StatusGetTstart* for more information on the *tstart* value.

Parameters

status	structure containing current simulation info
tstart_ptr	output, current time
tstop_ptr	output, next time value to evolve towards

10.9.2.28 braid_StatusGetTstop()

```
braid_Int braid_StatusGetTstop (
            braid_Status status,
            braid_Real * tstop_ptr )
```

Return the time value to the right of the current time value from the Status structure.

Parameters

status	structure containing current simulation info
tstop_ptr	output, next time value to evolve towards

10.9.2.29 braid_StatusGetWrapperTest()

```
braid_Int braid_StatusGetWrapperTest (
            braid_Status status,
            braid_Int * wtest_ptr )
```

Return whether this is a wrapper test or an XBraid run

Parameters

status	structure containing current simulation info
wtest_ptr	output, =1 if this is a wrapper test, =0 if XBraid run

10.9.2.30 braid_StatusSetOldFineTolx()

```
braid_Int braid_StatusSetOldFineTolx (
            braid_Status status,
            braid_Real old_fine_tolx )
```

Set *old_fine_tolx*, available for retrieval through *braid_StatusGetOldFineTolx* This is used especially by *braid_Get↩
SpatialAccuracy

Parameters

status	structure containing current simulation info
old_fine_tolx	input, the last used fine_tolx

10.9.2.31 braid_StatusSetRefinementDtValues()

```
braid_Int braid_StatusSetRefinementDtValues (
            braid_Status status,
            braid_Real rfactor,
            braid_Real * dtarray )
```

Set time step sizes for refining the time interval non-uniformly.

Parameters

status	structure containing current simulation info
rfactor	input, number of subintervals
dtarray	input, array of dt values for non-uniform refinement

10.9.2.32 braid_StatusSetRFactor()

```
braid_Int braid_StatusSetRFactor (
              braid_Status status,
              braid_Real rfactor )
```

Set the rfactor, a desired refinement factor for this interval. rfactor=1 indicates no refinement, otherwise, this inteval is subdivided rfactor times (uniform refinement).

Parameters

status	structure containing current simulation info
rfactor	input, user-determined desired rfactor

10.9.2.33 braid_StatusSetRSpace()

```
braid_Int braid_StatusSetRSpace (
              braid_Status status,
              braid_Real r_space )
```

Set the r_space flag. When set = 1, spatial coarsening will be called, for all local time points, following the completion of the current iteration, provided rfactors are not set at any global time point. This allows for spatial refinment without temporal refinment

Parameters

status	structure containing current simulation info
r_space	input, if 1, call spatial refinement on finest grid after this iter

10.9.2.34 braid_StatusSetSize()

```
braid_Int braid_StatusSetSize (
              braid_Status status,
              braid_Real size )
```

Set the size of the buffer. If set by user, the send buffer will be "size" bytes in length. If not, BufSize is used.

Parameters

status	structure containing current simulation info
size	input, size of the send buffer

10.9.2.35 braid_StatusSetTightFineTolx()

```
braid_Int braid_StatusSetTightFineTolx (
              braid_Status status,
```

```
braid_Real tight_fine_tolx )
```

Set *tight_fine_tolx*, boolean variable indicating whether the tightest tolerance has been used for spatial solves (implicit schemes). This value must be 1 in order for XBraid to halt (unless maxiter is reached)

Parameters

status	structure containing current simulation info
tight_fine_tolx	input, boolean indicating whether the tight tolx has been used

10.10 Inherited XBraid status routines

Functions

- braid_Int braid_AccessStatusGetT (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_AccessStatusGetTIndex (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetIter (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetLevel (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNLevels (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNRefine (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNTPoints (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetResidual (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_AccessStatusGetDone (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetTILD (braid_AccessStatus s, braid_Real ∗v1, braid_Int ∗v2, braid_Int ∗v3, braid_Int ∗v4)
- braid_Int braid_AccessStatusGetWrapperTest (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetCallingFunction (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetSingleErrorEstAccess (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_SyncStatusGetTIUL (braid_SyncStatus s, braid_Int ∗v1, braid_Int ∗v2, braid_Int v3)
- braid_Int braid_SyncStatusGetTimeValues (braid_SyncStatus s, braid_Real ∗∗v1, braid_Int v2, braid_Int v3, braid_Int v4)
- braid_Int braid_SyncStatusGetIter (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetLevel (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNLevels (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNRefine (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNTPoints (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetDone (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetCallingFunction (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNumErrorEst (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetAllErrorEst (braid_SyncStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetT (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetTIndex (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetIter (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetLevel (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNLevels (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNRefine (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNTPoints (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetCTprior (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetCTstop (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetFTprior (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetFTstop (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetTpriorTstop (braid_CoarsenRefStatus s, braid_Real ∗v1, braid_Real ∗v2, braid_Real ∗v3, braid_Real ∗v4, braid_Real ∗v5)
- braid_Int braid_StepStatusGetT (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusGetTIndex (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetIter (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetLevel (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNLevels (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNRefine (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNTPoints (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetTstop (braid_StepStatus s, braid_Real ∗v1)

- braid_Int braid_StepStatusGetTstartTstop (braid_StepStatus s, braid_Real ∗v1, braid_Real ∗v2)
- braid_Int braid_StepStatusGetTol (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusGetRNorms (braid_StepStatus s, braid_Int ∗v1, braid_Real ∗v2)
- braid_Int braid_StepStatusGetOldFineTolx (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusSetOldFineTolx (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetTightFineTolx (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetRFactor (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetRSpace (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusGetSingleErrorEstStep (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_BufferStatusGetMessageType (braid_BufferStatus s, braid_Int ∗v1)
- braid_Int braid_BufferStatusSetSize (braid_BufferStatus s, braid_Real v1)
- braid_Int braid_ObjectiveStatusGetT (braid_ObjectiveStatus s, braid_Real ∗v1)
- braid_Int braid_ObjectiveStatusGetTIndex (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetIter (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetLevel (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNLevels (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNRefine (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNTPoints (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetTol (braid_ObjectiveStatus s, braid_Real ∗v1)

10.10.1 Detailed Description

These are the 'inherited' Status Get/Set functions. See the *XBraid status routines* section for the description of each function. For example, for braid_StepStatusGetT(...), you would look up braid_StatusGetT(...)

10.10.2 Function Documentation

10.10.2.1 braid_AccessStatusGetCallingFunction()

```
braid_Int braid_AccessStatusGetCallingFunction (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.2 braid_AccessStatusGetDone()

```
braid_Int braid_AccessStatusGetDone (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.3 braid_AccessStatusGetIter()

```
braid_Int braid_AccessStatusGetIter (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.4 braid_AccessStatusGetLevel()

```
braid_Int braid_AccessStatusGetLevel (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.5 braid_AccessStatusGetNLevels()

```
braid_Int braid_AccessStatusGetNLevels (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.6 braid_AccessStatusGetNRefine()

```
braid_Int braid_AccessStatusGetNRefine (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.7 braid_AccessStatusGetNTPoints()

```
braid_Int braid_AccessStatusGetNTPoints (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.8 braid_AccessStatusGetResidual()

```
braid_Int braid_AccessStatusGetResidual (
            braid_AccessStatus s,
            braid_Real * v1 )
```

10.10.2.9 braid_AccessStatusGetSingleErrorEstAccess()

```
braid_Int braid_AccessStatusGetSingleErrorEstAccess (
            braid_AccessStatus s,
            braid_Real * v1 )
```

10.10.2.10 braid_AccessStatusGetT()

```
braid_Int braid_AccessStatusGetT (
            braid_AccessStatus s,
            braid_Real * v1 )
```

10.10.2.11 braid_AccessStatusGetTILD()

```
braid_Int braid_AccessStatusGetTILD (
            braid_AccessStatus s,
```

```
            braid_Real * v1,
            braid_Int * v2,
            braid_Int * v3,
            braid_Int * v4 )
```

10.10.2.12 braid_AccessStatusGetTIndex()

```
braid_Int braid_AccessStatusGetTIndex (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.13 braid_AccessStatusGetWrapperTest()

```
braid_Int braid_AccessStatusGetWrapperTest (
            braid_AccessStatus s,
            braid_Int * v1 )
```

10.10.2.14 braid_BufferStatusGetMessageType()

```
braid_Int braid_BufferStatusGetMessageType (
            braid_BufferStatus s,
            braid_Int * v1 )
```

10.10.2.15 braid_BufferStatusSetSize()

```
braid_Int braid_BufferStatusSetSize (
            braid_BufferStatus s,
            braid_Real v1 )
```

10.10.2.16 braid_CoarsenRefStatusGetCTprior()

```
braid_Int braid_CoarsenRefStatusGetCTprior (
            braid_CoarsenRefStatus s,
            braid_Real * v1 )
```

10.10.2.17 braid_CoarsenRefStatusGetCTstop()

```
braid_Int braid_CoarsenRefStatusGetCTstop (
            braid_CoarsenRefStatus s,
            braid_Real * v1 )
```

10.10.2.18 braid_CoarsenRefStatusGetFTprior()

```
braid_Int braid_CoarsenRefStatusGetFTprior (
            braid_CoarsenRefStatus s,
            braid_Real * v1 )
```

10.10.2.19 braid_CoarsenRefStatusGetFTstop()

```
braid_Int braid_CoarsenRefStatusGetFTstop (
            braid_CoarsenRefStatus s,
            braid_Real * v1 )
```

10.10.2.20 braid_CoarsenRefStatusGetIter()

```
braid_Int braid_CoarsenRefStatusGetIter (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.21 braid_CoarsenRefStatusGetLevel()

```
braid_Int braid_CoarsenRefStatusGetLevel (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.22 braid_CoarsenRefStatusGetNLevels()

```
braid_Int braid_CoarsenRefStatusGetNLevels (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.23 braid_CoarsenRefStatusGetNRefine()

```
braid_Int braid_CoarsenRefStatusGetNRefine (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.24 braid_CoarsenRefStatusGetNTPoints()

```
braid_Int braid_CoarsenRefStatusGetNTPoints (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.25 braid_CoarsenRefStatusGetT()

```
braid_Int braid_CoarsenRefStatusGetT (
            braid_CoarsenRefStatus s,
            braid_Real * v1 )
```

10.10.2.26 braid_CoarsenRefStatusGetTIndex()

```
braid_Int braid_CoarsenRefStatusGetTIndex (
            braid_CoarsenRefStatus s,
            braid_Int * v1 )
```

10.10.2.27 braid_CoarsenRefStatusGetTpriorTstop()

```
braid_Int braid_CoarsenRefStatusGetTpriorTstop (
            braid_CoarsenRefStatus s,
            braid_Real * v1,
            braid_Real * v2,
            braid_Real * v3,
            braid_Real * v4,
            braid_Real * v5 )
```

10.10.2.28 braid_ObjectiveStatusGetIter()

```
braid_Int braid_ObjectiveStatusGetIter (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.29 braid_ObjectiveStatusGetLevel()

```
braid_Int braid_ObjectiveStatusGetLevel (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.30 braid_ObjectiveStatusGetNLevels()

```
braid_Int braid_ObjectiveStatusGetNLevels (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.31 braid_ObjectiveStatusGetNRefine()

```
braid_Int braid_ObjectiveStatusGetNRefine (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.32 braid_ObjectiveStatusGetNTPoints()

```
braid_Int braid_ObjectiveStatusGetNTPoints (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.33 braid_ObjectiveStatusGetT()

```
braid_Int braid_ObjectiveStatusGetT (
            braid_ObjectiveStatus s,
            braid_Real * v1 )
```

10.10.2.34 braid_ObjectiveStatusGetTIndex()

```
braid_Int braid_ObjectiveStatusGetTIndex (
            braid_ObjectiveStatus s,
            braid_Int * v1 )
```

10.10.2.35 braid_ObjectiveStatusGetTol()

```
braid_Int braid_ObjectiveStatusGetTol (
            braid_ObjectiveStatus s,
            braid_Real * v1 )
```

10.10.2.36 braid_StepStatusGetIter()

```
braid_Int braid_StepStatusGetIter (
            braid_StepStatus s,
            braid_Int * v1 )
```

10.10.2.37 braid_StepStatusGetLevel()

```
braid_Int braid_StepStatusGetLevel (
            braid_StepStatus s,
            braid_Int * v1 )
```

10.10.2.38 braid_StepStatusGetNLevels()

```
braid_Int braid_StepStatusGetNLevels (
            braid_StepStatus s,
            braid_Int * v1 )
```

10.10.2.39 braid_StepStatusGetNRefine()

```
braid_Int braid_StepStatusGetNRefine (
            braid_StepStatus s,
            braid_Int * v1 )
```

10.10.2.40 braid_StepStatusGetNTPoints()

```
braid_Int braid_StepStatusGetNTPoints (
            braid_StepStatus s,
```

braid_Int * v1)

10.10.2.41 braid_StepStatusGetOldFineTolx()

```
braid_Int braid_StepStatusGetOldFineTolx (
            braid_StepStatus s,
            braid_Real * v1 )
```

10.10.2.42 braid_StepStatusGetRNorms()

```
braid_Int braid_StepStatusGetRNorms (
            braid_StepStatus s,
            braid_Int * v1,
            braid_Real * v2 )
```

10.10.2.43 braid_StepStatusGetSingleErrorEstStep()

```
braid_Int braid_StepStatusGetSingleErrorEstStep (
            braid_StepStatus s,
            braid_Real * v1 )
```

10.10.2.44 braid_StepStatusGetT()

```
braid_Int braid_StepStatusGetT (
            braid_StepStatus s,
            braid_Real * v1 )
```

10.10.2.45 braid_StepStatusGetTIndex()

```
braid_Int braid_StepStatusGetTIndex (
            braid_StepStatus s,
            braid_Int * v1 )
```

10.10.2.46 braid_StepStatusGetTol()

```
braid_Int braid_StepStatusGetTol (
            braid_StepStatus s,
            braid_Real * v1 )
```

10.10.2.47 braid_StepStatusGetTstartTstop()

```
braid_Int braid_StepStatusGetTstartTstop (
            braid_StepStatus s,
            braid_Real * v1,
            braid_Real * v2 )
```

10.10.2.48 braid_StepStatusGetTstop()

```
braid_Int braid_StepStatusGetTstop (
            braid_StepStatus s,
            braid_Real * v1 )
```

10.10.2.49 braid_StepStatusSetOldFineTolx()

```
braid_Int braid_StepStatusSetOldFineTolx (
            braid_StepStatus s,
            braid_Real v1 )
```

10.10.2.50 braid_StepStatusSetRFactor()

```
braid_Int braid_StepStatusSetRFactor (
            braid_StepStatus s,
            braid_Real v1 )
```

10.10.2.51 braid_StepStatusSetRSpace()

```
braid_Int braid_StepStatusSetRSpace (
            braid_StepStatus s,
            braid_Real v1 )
```

10.10.2.52 braid_StepStatusSetTightFineTolx()

```
braid_Int braid_StepStatusSetTightFineTolx (
            braid_StepStatus s,
            braid_Real v1 )
```

10.10.2.53 braid_SyncStatusGetAllErrorEst()

```
braid_Int braid_SyncStatusGetAllErrorEst (
            braid_SyncStatus s,
            braid_Real * v1 )
```

10.10.2.54 braid_SyncStatusGetCallingFunction()

```
braid_Int braid_SyncStatusGetCallingFunction (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.55 braid_SyncStatusGetDone()

```
braid_Int braid_SyncStatusGetDone (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.56 braid_SyncStatusGetIter()

```
braid_Int braid_SyncStatusGetIter (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.57 braid_SyncStatusGetLevel()

```
braid_Int braid_SyncStatusGetLevel (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.58 braid_SyncStatusGetNLevels()

```
braid_Int braid_SyncStatusGetNLevels (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.59 braid_SyncStatusGetNRefine()

```
braid_Int braid_SyncStatusGetNRefine (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.60 braid_SyncStatusGetNTPoints()

```
braid_Int braid_SyncStatusGetNTPoints (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.61 braid_SyncStatusGetNumErrorEst()

```
braid_Int braid_SyncStatusGetNumErrorEst (
            braid_SyncStatus s,
            braid_Int * v1 )
```

10.10.2.62 braid_SyncStatusGetTimeValues()

```
braid_Int braid_SyncStatusGetTimeValues (
            braid_SyncStatus s,
```

```
            braid_Real ** v1,
            braid_Int v2,
            braid_Int v3,
            braid_Int v4 )
```

10.10.2.63 braid_SyncStatusGetTIUL()

```
braid_Int braid_SyncStatusGetTIUL (
            braid_SyncStatus s,
            braid_Int * v1,
            braid_Int * v2,
            braid_Int v3 )
```

10.11 XBraid status macros

Macros

- #define braid_ASCaller_FInterp 0
- #define braid_ASCaller_FRestrict 1
- #define braid_ASCaller_FRefine 2
- #define braid_ASCaller_FAccess 3
- #define braid_ASCaller_FRefine_AfterInitHier 4
- #define braid_ASCaller_Drive_TopCycle 5

10.11.1 Detailed Description

Macros defining Status values that the user can obtain during runtime, which will tell the user where in Braid the current cycle is, e.g. in the FInterp function.

10.11.2 Macro Definition Documentation

10.11.2.1 braid_ASCaller_Drive_TopCycle

```
#define braid_ASCaller_Drive_TopCycle 5
```

When CallingFunction equals 5, Braid is at the top of the cycle

10.11.2.2 braid_ASCaller_FAccess

```
#define braid_ASCaller_FAccess 3
```

When CallingFunction equals 0, Braid is in FAccess

10.11.2.3 braid_ASCaller_FInterp

```
#define braid_ASCaller_FInterp 0
```

When CallingFunction equals 0, Braid is in FInterp

10.11.2.4 braid_ASCaller_FRefine

```
#define braid_ASCaller_FRefine 2
```

When CallingFunction equals 0, Braid is in FRefine

10.11.2.5 braid_ASCaller_FRefine_AfterInitHier

```
#define braid_ASCaller_FRefine_AfterInitHier 4
```

When CallingFunction equals 4, Braid is inside FRefine after the new finest level has been initialized

10.11.2.6 braid_ASCaller_FRestrict

```
#define braid_ASCaller_FRestrict 1
```

When CallingFunction equals 0, Braid is in FRestrict

10.12 XBraid test routines

Functions

- braid_Int braid_TestInitAccess (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestSpatialNorm (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnSCoarsen coarsen, braid_Pt←FcnSRefine refine)
- braid_Int braid_TestResidual (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real dt, braid←_PtFcnInit myinit, braid_PtFcnAccess myaccess, braid_PtFcnFree myfree, braid_PtFcnClone clone, braid_Pt←FcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnResidual residual, braid_PtFcnStep step)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_←Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_Pt←FcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnSCoarsen coarsen, braid_PtFcnSRefine refine, braid_PtFcnResidual residual, braid_←PtFcnStep step)

10.12.1 Detailed Description

These are sanity check routines to help a user test their XBraid code.

10.12.2 Function Documentation

10.12.2.1 braid_TestAll()

```
braid_Int braid_TestAll (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_Real fdt,
            braid_Real cdt,
            braid_PtFcnInit init,
            braid_PtFcnFree free,
            braid_PtFcnClone clone,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm,
            braid_PtFcnBufSize bufsize,
            braid_PtFcnBufPack bufpack,
```

```
            braid_PtFcnBufUnpack bufunpack,
            braid_PtFcnSCoarsen coarsen,
            braid_PtFcnSRefine refine,
            braid_PtFcnResidual residual,
            braid_PtFcnStep step )
```

Runs all of the individual braid_Test∗ routines

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors with
fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer into a braid_Vector
coarsen	Spatially coarsen a vector. If NULL, test is skipped.
refine	Spatially refine a vector. If NULL, test is skipped.
residual	Compute a residual given two consectuive braid_Vectors
step	Compute a time step with a braid_Vector

10.12.2.2 braid_TestBuf()

```
braid_Int braid_TestBuf (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_PtFcnInit init,
            braid_PtFcnFree free,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm,
            braid_PtFcnBufSize bufsize,
            braid_PtFcnBufPack bufpack,
```

```
            braid_PtFcnBufUnpack bufunpack )
```

Test the BufPack, BufUnpack and BufSize functions.
A vector is initialized at time *t*, packed into a buffer, then unpacked from a buffer. The unpacked result must equal the original vector.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Buffer routines (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer containing one braid_Vector

10.12.2.3 braid_TestClone()

```
braid_Int braid_TestClone (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_PtFcnInit init,
            braid_PtFcnAccess access,
            braid_PtFcnFree free,
            braid_PtFcnClone clone )
```

Test the clone function.
A vector is initialized at time *t*, cloned, and both vectors are written. Then both vectors are free-d. The user is to check (via the access function) to see if it is identical.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test clone with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid

Parameters

access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector

10.12.2.4 braid_TestCoarsenRefine()

```
braid_Int braid_TestCoarsenRefine (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_Real fdt,
            braid_Real cdt,
            braid_PtFcnInit init,
            braid_PtFcnAccess access,
            braid_PtFcnFree free,
            braid_PtFcnClone clone,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm,
            braid_PtFcnSCoarsen coarsen,
            braid_PtFcnSRefine refine )
```

Test the Coarsen and Refine functions.
A vector is initialized at time *t*, and various sanity checks on the spatial coarsening and refinement routines are run.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors
fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
coarsen	Spatially coarsen a vector
refine	Spatially refine a vector

10.12.2.5 braid_TestInitAccess()

```
braid_Int braid_TestInitAccess (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_PtFcnInit init,
            braid_PtFcnAccess access,
            braid_PtFcnFree free )
```

Test the init, access and free functions.
A vector is initialized at time *t*, written, and then free-d

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test init with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector

10.12.2.6 braid_TestResidual()

```
braid_Int braid_TestResidual (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_Real dt,
            braid_PtFcnInit myinit,
            braid_PtFcnAccess myaccess,
            braid_PtFcnFree myfree,
            braid_PtFcnClone clone,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm,
            braid_PtFcnResidual residual,
            braid_PtFcnStep step )
```

Test compatibility of the Step and Residual functions.
A vector is initialized at time *t*, step is called with *dt*, followed by an evaluation of residual, to test the condition fstop - residual(step(u, fstop), u) approx. 0

- Check the log messages to determine if test passed. The result should approximately be zero. The more accurate the solution for *u* is computed in step, the closer the result will be to 0.
- The residual is also written to file

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors
dt	Time step value to use in step
myinit	Initialize a braid_Vector on finest temporal grid
myaccess	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
myfree	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space
residual	Compute a residual given two consectuive braid_Vectors
step	Compute a time step with a braid_Vector

10.12.2.7 braid_TestSpatialNorm()

```
braid_Int braid_TestSpatialNorm (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_PtFcnInit init,
            braid_PtFcnFree free,
            braid_PtFcnClone clone,
            braid_PtFcnSum sum,
            braid_PtFcnSpatialNorm spatialnorm )
```

Test the spatialnorm function.
A vector is initialized at time *t* and then cloned. Various norm evaluations like || 3 v || / || v || with known output are then done.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test SpatialNorm with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector

Parameters

sum	Compute vector sum of two braid_Vectors
spatialnorm	Compute norm of a braid_Vector, this is a norm only over space

10.12.2.8 braid_TestSum()

```
braid_Int braid_TestSum (
            braid_App app,
            MPI_Comm comm_x,
            FILE * fp,
            braid_Real t,
            braid_PtFcnInit init,
            braid_PtFcnAccess access,
            braid_PtFcnFree free,
            braid_PtFcnClone clone,
            braid_PtFcnSum sum )
```

Test the sum function.

A vector is initialized at time *t*, cloned, and then these two vectors are summed a few times, with the results written. The vectors are then free-d. The user is to check (via the access function) that the output matches the sum of the two original vectors.

Parameters

app	User defined App structure
comm↔_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Sum with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
access	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors

11 File Documentation

11.1 braid.h File Reference

Macros

- #define braid_FMANGLE 1
- #define braid_Fortran_SpatialCoarsen 0
- #define braid_Fortran_Residual 1
- #define braid_Fortran_TimeGrid 1
- #define braid_Fortran_Sync 1
- #define braid_INVALID_RNORM -1
- #define braid_ERROR_GENERIC 1 /∗ generic error ∗/
- #define braid_ERROR_MEMORY 2 /∗ unable to allocate memory ∗/
- #define braid_ERROR_ARG 4 /∗ argument error ∗/
- #define braid_RAND_MAX 32768

Typedefs

- typedef struct _braid_App_struct ∗ braid_App
- typedef struct _braid_Vector_struct ∗ braid_Vector
- typedef braid_Int(∗ braid_PtFcnStep) (braid_App app, braid_Vector ustop, braid_Vector fstop, braid_Vector u, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnInit) (braid_App app, braid_Real t, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnClone) (braid_App app, braid_Vector u, braid_Vector ∗v_ptr)
- typedef braid_Int(∗ braid_PtFcnFree) (braid_App app, braid_Vector u)
- typedef braid_Int(∗ braid_PtFcnSum) (braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid↩_Vector y)
- typedef braid_Int(∗ braid_PtFcnSpatialNorm) (braid_App app, braid_Vector u, braid_Real ∗norm_ptr)
- typedef braid_Int(∗ braid_PtFcnAccess) (braid_App app, braid_Vector u, braid_AccessStatus status)
- typedef braid_Int(∗ braid_PtFcnSync) (braid_App app, braid_SyncStatus status)
- typedef braid_Int(∗ braid_PtFcnBufSize) (braid_App app, braid_Int ∗size_ptr, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnBufPack) (braid_App app, braid_Vector u, void ∗buffer, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnBufUnpack) (braid_App app, void ∗buffer, braid_Vector ∗u_ptr, braid_BufferStatus status)
- typedef braid_Int(∗ braid_PtFcnResidual) (braid_App app, braid_Vector ustop, braid_Vector r, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnSCoarsen) (braid_App app, braid_Vector fu, braid_Vector ∗cu_ptr, braid_↩CoarsenRefStatus status)
- typedef braid_Int(∗ braid_PtFcnSRefine) (braid_App app, braid_Vector cu, braid_Vector ∗fu_ptr, braid_Coarsen↩RefStatus status)
- typedef braid_Int(∗ braid_PtFcnSInit) (braid_App app, braid_Real t, braid_Vector ∗u_ptr)
- typedef braid_Int(∗ braid_PtFcnSClone) (braid_App app, braid_Vector u, braid_Vector ∗v_ptr)
- typedef braid_Int(∗ braid_PtFcnSFree) (braid_App app, braid_Vector u)
- typedef braid_Int(∗ braid_PtFcnTimeGrid) (braid_App app, braid_Real ∗ta, braid_Int ∗ilower, braid_Int ∗iupper)
- typedef braid_Int(∗ braid_PtFcnObjectiveT) (braid_App app, braid_Vector u, braid_ObjectiveStatus ostatus, braid_Real ∗objectiveT_ptr)
- typedef braid_Int(∗ braid_PtFcnObjectiveTDiff) (braid_App app, braid_Vector u, braid_Vector u_bar, braid_Real F_bar, braid_ObjectiveStatus ostatus)
- typedef braid_Int(∗ braid_PtFcnPostprocessObjective) (braid_App app, braid_Real sum_obj, braid_Real ∗postprocess_ptr)

- typedef braid_Int(∗ braid_PtFcnPostprocessObjective_diff) (braid_App app, braid_Real sum_obj, braid_Real ∗F←_bar_ptr)
- typedef braid_Int(∗ braid_PtFcnStepDiff) (braid_App app, braid_Vector ustop, braid_Vector u, braid_Vector ustop_bar, braid_Vector u_bar, braid_StepStatus status)
- typedef braid_Int(∗ braid_PtFcnResetGradient) (braid_App app)
- typedef struct _braid_Core_struct ∗ braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_←Int ntime, braid_App app, braid_PtFcnStep step, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnAccess access, braid_PtFcnBuf←Size bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core ∗core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_WriteConvHistory (braid_Core core, const char ∗filename)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetIncrMaxLevels (braid_Core core)
- braid_Int braid_SetSkip (braid_Core core, braid_Int skip)
- braid_Int braid_SetRefine (braid_Core core, braid_Int refine)
- braid_Int braid_SetMaxRefinements (braid_Core core, braid_Int max_refinements)
- braid_Int braid_SetTPointsCutoff (braid_Core core, braid_Int tpoints_cutoff)
- braid_Int braid_SetMinCoarse (braid_Core core, braid_Int min_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCRelaxWt (braid_Core core, braid_Int level, braid_Real Cwt)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetNFMG (braid_Core core, braid_Int k)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)
- braid_Int braid_SetStorage (braid_Core core, braid_Int storage)
- braid_Int braid_SetTemporalNorm (braid_Core core, braid_Int tnorm)
- braid_Int braid_SetResidual (braid_Core core, braid_PtFcnResidual residual)
- braid_Int braid_SetFullRNormRes (braid_Core core, braid_PtFcnResidual residual)
- braid_Int braid_SetTimeGrid (braid_Core core, braid_PtFcnTimeGrid tgrid)
- braid_Int braid_SetPeriodic (braid_Core core, braid_Int periodic)
- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnSCoarsen scoarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnSRefine srefine)
- braid_Int braid_SetSync (braid_Core core, braid_PtFcnSync sync)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)
- braid_Int braid_SetFileIOLevel (braid_Core core, braid_Int io_level)
- braid_Int braid_SetPrintFile (braid_Core core, const char ∗printfile_name)
- braid_Int braid_SetDefaultPrintFile (braid_Core core)
- braid_Int braid_SetAccessLevel (braid_Core core, braid_Int access_level)
- braid_Int braid_SplitCommworld (const MPI_Comm ∗comm_world, braid_Int px, MPI_Comm ∗comm_x, MPI_←Comm ∗comm_t)
- braid_Int braid_SetShell (braid_Core core, braid_PtFcnSInit sinit, braid_PtFcnSClone sclone, braid_PtFcnSFree sfree)

- braid_Int braid_GetNumIter (braid_Core core, braid_Int ∗niter_ptr)
- braid_Int braid_GetRNorms (braid_Core core, braid_Int ∗nrequest_ptr, braid_Real ∗rnorms)
- braid_Int braid_GetNLevels (braid_Core core, braid_Int ∗nlevels_ptr)
- braid_Int braid_GetSpatialAccuracy (braid_StepStatus status, braid_Real loose_tol, braid_Real tight_tol, braid_Real ∗tol_ptr)
- braid_Int braid_SetSeqSoln (braid_Core core, braid_Int seq_soln)
- braid_Int braid_GetMyID (braid_Core core, braid_Int ∗myid_ptr)
- braid_Int braid_Rand (void)
- braid_Int braid_InitAdjoint (braid_PtFcnObjectiveT objectiveT, braid_PtFcnObjectiveTDiff objectiveT_diff, braid_PtFcnStepDiff step_diff, braid_PtFcnResetGradient reset_gradient, braid_Core ∗core_ptr)
- braid_Int braid_SetTStartObjective (braid_Core core, braid_Real tstart_obj)
- braid_Int braid_SetTStopObjective (braid_Core core, braid_Real tstop_obj)
- braid_Int braid_SetPostprocessObjective (braid_Core core, braid_PtFcnPostprocessObjective post_fcn)
- braid_Int braid_SetPostprocessObjective_diff (braid_Core core, braid_PtFcnPostprocessObjective_diff post_fcn_diff)
- braid_Int braid_SetAbsTolAdjoint (braid_Core core, braid_Real tol_adj)
- braid_Int braid_SetRelTolAdjoint (braid_Core core, braid_Real rtol_adj)
- braid_Int braid_SetObjectiveOnly (braid_Core core, braid_Int boolean)
- braid_Int braid_GetObjective (braid_Core core, braid_Real ∗objective_ptr)
- braid_Int braid_GetRNormAdjoint (braid_Core core, braid_Real ∗rnorm_adj)
- braid_Int braid_SetRichardsonEstimation (braid_Core core, braid_Int est_error, braid_Int richardson, braid_Int local_order)

11.1.1 Detailed Description

Define headers for user-interface routines.

This file contains user-routines used to allow the user to initialize, run and get and set options for a XBraid solver.

11.2 braid_defs.h File Reference

Macros

- #define braid_Int_Max INT_MAX;
- #define braid_Int_Min INT_MIN;
- #define braid_MPI_REAL MPI_DOUBLE
- #define braid_MPI_INT MPI_INT

Typedefs

- typedef int braid_Int
- typedef double braid_Real

11.2.1 Detailed Description

Definitions of braid types, error flags, etc...

11.2.2 Macro Definition Documentation

11.2.2.1 braid_Int_Max

```
#define braid_Int_Max INT_MAX;
```

11.2.2.2 braid_Int_Min

```
#define braid_Int_Min INT_MIN;
```

11.2.2.3 braid_MPI_INT

```
#define braid_MPI_INT MPI_INT
```

11.2.2.4 braid_MPI_REAL

```
#define braid_MPI_REAL MPI_DOUBLE
```

11.2.3 Typedef Documentation

11.2.3.1 braid_Int

```
typedef int braid_Int
```

Defines integer type

11.2.3.2 braid_Real

```
typedef double braid_Real
```

Defines floating point type

11.3 braid_status.h File Reference

Macros

- #define ACCESSOR_HEADER_GET1(stype, param, vtype1) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1);
- #define ACCESSOR_HEADER_GET1_IN3(stype, param, vtype1, vtype2, vtype3, vtype4) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1, braid_##vtype2 v2, braid_##vtype3 v3, braid_##vtype4 v4);
- #define ACCESSOR_HEADER_GET2(stype, param, vtype1, vtype2) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1, braid_##vtype2 *v2);
- #define ACCESSOR_HEADER_GET2_IN1(stype, param, vtype1, vtype2, vtype3) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 v3);
- #define ACCESSOR_HEADER_GET3(stype, param, vtype1, vtype2, vtype3) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 *v3);
- #define ACCESSOR_HEADER_GET4(stype, param, vtype1, vtype2, vtype3, vtype4) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 *v3, braid_##vtype4 *v4);

- #define ACCESSOR_HEADER_GET5(stype, param, vtype1, vtype2, vtype3, vtype4, vtype5) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_##vtype1 ∗v1, braid_##vtype2 ∗v2, braid_##vtype3 ∗v3, braid_##vtype4 ∗v4, braid_##vtype5 ∗v5);
- #define ACCESSOR_HEADER_SET1(stype, param, vtype1) braid_Int braid_##stype##StatusSet##param(braid_##stype##Status s, braid_##vtype1 v1);
- #define braid_ASCaller_FInterp 0
- #define braid_ASCaller_FRestrict 1
- #define braid_ASCaller_FRefine 2
- #define braid_ASCaller_FAccess 3
- #define braid_ASCaller_FRefine_AfterInitHier 4
- #define braid_ASCaller_Drive_TopCycle 5

Typedefs

- typedef struct _braid_Status_struct ∗ braid_Status
- typedef struct _braid_AccessStatus_struct ∗ braid_AccessStatus
- typedef struct _braid_SyncStatus_struct ∗ braid_SyncStatus
- typedef struct _braid_StepStatus_struct ∗ braid_StepStatus
- typedef struct _braid_CoarsenRefStatus_struct ∗ braid_CoarsenRefStatus
- typedef struct _braid_BufferStatus_struct ∗ braid_BufferStatus
- typedef struct _braid_ObjectiveStatus_struct ∗ braid_ObjectiveStatus

Functions

- braid_Int braid_StatusGetT (braid_Status status, braid_Real ∗t_ptr)
- braid_Int braid_StatusGetTIndex (braid_Status status, braid_Int ∗idx_ptr)
- braid_Int braid_StatusGetIter (braid_Status status, braid_Int ∗iter_ptr)
- braid_Int braid_StatusGetLevel (braid_Status status, braid_Int ∗level_ptr)
- braid_Int braid_StatusGetNLevels (braid_Status status, braid_Int ∗nlevels_ptr)
- braid_Int braid_StatusGetNRefine (braid_Status status, braid_Int ∗nrefine_ptr)
- braid_Int braid_StatusGetNTPoints (braid_Status status, braid_Int ∗ntpoints_ptr)
- braid_Int braid_StatusGetResidual (braid_Status status, braid_Real ∗rnorm_ptr)
- braid_Int braid_StatusGetDone (braid_Status status, braid_Int ∗done_ptr)
- braid_Int braid_StatusGetTIUL (braid_Status status, braid_Int ∗iloc_upper, braid_Int ∗iloc_lower, braid_Int level)
- braid_Int braid_StatusGetTimeValues (braid_Status status, braid_Real ∗∗tvalues_ptr, braid_Int i_upper, braid_Int i_lower, braid_Int level)
- braid_Int braid_StatusGetTILD (braid_Status status, braid_Real ∗t_ptr, braid_Int ∗iter_ptr, braid_Int ∗level_ptr, braid_Int ∗done_ptr)
- braid_Int braid_StatusGetWrapperTest (braid_Status status, braid_Int ∗wtest_ptr)
- braid_Int braid_StatusGetCallingFunction (braid_Status status, braid_Int ∗cfunction_ptr)
- braid_Int braid_StatusGetCTprior (braid_Status status, braid_Real ∗ctprior_ptr)
- braid_Int braid_StatusGetCTstop (braid_Status status, braid_Real ∗ctstop_ptr)
- braid_Int braid_StatusGetFTprior (braid_Status status, braid_Real ∗ftprior_ptr)
- braid_Int braid_StatusGetFTstop (braid_Status status, braid_Real ∗ftstop_ptr)
- braid_Int braid_StatusGetTpriorTstop (braid_Status status, braid_Real ∗t_ptr, braid_Real ∗ftprior_ptr, braid_Real ∗ftstop_ptr, braid_Real ∗ctprior_ptr, braid_Real ∗ctstop_ptr)
- braid_Int braid_StatusGetTstop (braid_Status status, braid_Real ∗tstop_ptr)
- braid_Int braid_StatusGetTstartTstop (braid_Status status, braid_Real ∗tstart_ptr, braid_Real ∗tstop_ptr)
- braid_Int braid_StatusGetTol (braid_Status status, braid_Real ∗tol_ptr)
- braid_Int braid_StatusGetRNorms (braid_Status status, braid_Int ∗nrequest_ptr, braid_Real ∗rnorms_ptr)

- braid_Int braid_StatusGetOldFineTolx (braid_Status status, braid_Real ∗old_fine_tolx_ptr)
- braid_Int braid_StatusSetOldFineTolx (braid_Status status, braid_Real old_fine_tolx)
- braid_Int braid_StatusSetTightFineTolx (braid_Status status, braid_Real tight_fine_tolx)
- braid_Int braid_StatusSetRFactor (braid_Status status, braid_Real rfactor)
- braid_Int braid_StatusSetRefinementDtValues (braid_Status status, braid_Real rfactor, braid_Real ∗dtarray)
- braid_Int braid_StatusSetRSpace (braid_Status status, braid_Real r_space)
- braid_Int braid_StatusGetMessageType (braid_Status status, braid_Int ∗messagetype_ptr)
- braid_Int braid_StatusSetSize (braid_Status status, braid_Real size)
- braid_Int braid_StatusGetSingleErrorEstStep (braid_Status status, braid_Real ∗estimate)
- braid_Int braid_StatusGetSingleErrorEstAccess (braid_Status status, braid_Real ∗estimate)
- braid_Int braid_StatusGetNumErrorEst (braid_Status status, braid_Int ∗npoints)
- braid_Int braid_StatusGetAllErrorEst (braid_Status status, braid_Real ∗error_est)
- braid_Int braid_AccessStatusGetT (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_AccessStatusGetTIndex (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetIter (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetLevel (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNLevels (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNRefine (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetNTPoints (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetResidual (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_AccessStatusGetDone (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetTILD (braid_AccessStatus s, braid_Real ∗v1, braid_Int ∗v2, braid_Int ∗v3, braid_Int ∗v4)
- braid_Int braid_AccessStatusGetWrapperTest (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetCallingFunction (braid_AccessStatus s, braid_Int ∗v1)
- braid_Int braid_AccessStatusGetSingleErrorEstAccess (braid_AccessStatus s, braid_Real ∗v1)
- braid_Int braid_SyncStatusGetTIUL (braid_SyncStatus s, braid_Int ∗v1, braid_Int ∗v2, braid_Int v3)
- braid_Int braid_SyncStatusGetTimeValues (braid_SyncStatus s, braid_Real ∗∗v1, braid_Int v2, braid_Int v3, braid_Int v4)
- braid_Int braid_SyncStatusGetIter (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetLevel (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNLevels (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNRefine (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNTPoints (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetDone (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetCallingFunction (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetNumErrorEst (braid_SyncStatus s, braid_Int ∗v1)
- braid_Int braid_SyncStatusGetAllErrorEst (braid_SyncStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetT (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetTIndex (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetIter (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetLevel (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNLevels (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNRefine (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetNTPoints (braid_CoarsenRefStatus s, braid_Int ∗v1)
- braid_Int braid_CoarsenRefStatusGetCTprior (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetCTstop (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetFTprior (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetFTstop (braid_CoarsenRefStatus s, braid_Real ∗v1)
- braid_Int braid_CoarsenRefStatusGetTpriorTstop (braid_CoarsenRefStatus s, braid_Real ∗v1, braid_Real ∗v2, braid_Real ∗v3, braid_Real ∗v4, braid_Real ∗v5)

- braid_Int braid_StepStatusGetT (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusGetTIndex (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetIter (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetLevel (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNLevels (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNRefine (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetNTPoints (braid_StepStatus s, braid_Int ∗v1)
- braid_Int braid_StepStatusGetTstop (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusGetTstartTstop (braid_StepStatus s, braid_Real ∗v1, braid_Real ∗v2)
- braid_Int braid_StepStatusGetTol (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusGetRNorms (braid_StepStatus s, braid_Int ∗v1, braid_Real ∗v2)
- braid_Int braid_StepStatusGetOldFineTolx (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_StepStatusSetOldFineTolx (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetTightFineTolx (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetRFactor (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusSetRSpace (braid_StepStatus s, braid_Real v1)
- braid_Int braid_StepStatusGetSingleErrorEstStep (braid_StepStatus s, braid_Real ∗v1)
- braid_Int braid_BufferStatusGetMessageType (braid_BufferStatus s, braid_Int ∗v1)
- braid_Int braid_BufferStatusSetSize (braid_BufferStatus s, braid_Real v1)
- braid_Int braid_ObjectiveStatusGetT (braid_ObjectiveStatus s, braid_Real ∗v1)
- braid_Int braid_ObjectiveStatusGetTIndex (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetIter (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetLevel (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNLevels (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNRefine (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetNTPoints (braid_ObjectiveStatus s, braid_Int ∗v1)
- braid_Int braid_ObjectiveStatusGetTol (braid_ObjectiveStatus s, braid_Real ∗v1)

11.3.1 Detailed Description

Define headers for the user-interface with the XBraid status structures, allowing the user to get/set status structure values.

11.3.2 Macro Definition Documentation

11.3.2.1 ACCESSOR_HEADER_GET1

```
#define ACCESSOR_HEADER_GET1(
                stype,
                param,
                vtype1 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1);
```

Macros allowing for auto-generation of 'inherited' StatusGet functions

11.3.2.2 ACCESSOR_HEADER_GET1_IN3

```
#define ACCESSOR_HEADER_GET1_IN3(
              stype,
              param,
              vtype1,
              vtype2,
              vtype3,
              vtype4 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1, braid_##vtype2 v2, braid_##vtype3 v3, braid_##vtype4 v4);
```

11.3.2.3 ACCESSOR_HEADER_GET2

```
#define ACCESSOR_HEADER_GET2(
              stype,
              param,
              vtype1,
              vtype2 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1, braid_##vtype2 *v2);
```

11.3.2.4 ACCESSOR_HEADER_GET2_IN1

```
#define ACCESSOR_HEADER_GET2_IN1(
              stype,
              param,
              vtype1,
              vtype2,
              vtype3 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 v3);
```

11.3.2.5 ACCESSOR_HEADER_GET3

```
#define ACCESSOR_HEADER_GET3(
              stype,
              param,
              vtype1,
              vtype2,
              vtype3 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 *v3);
```

11.3.2.6 ACCESSOR_HEADER_GET4

```
#define ACCESSOR_HEADER_GET4(
              stype,
              param,
              vtype1,
              vtype2,
              vtype3,
              vtype4 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
```

```
##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 *v3, braid_##vtype4 *v4);
```

11.3.2.7 ACCESSOR_HEADER_GET5

```
#define ACCESSOR_HEADER_GET5(
            stype,
            param,
            vtype1,
            vtype2,
            vtype3,
            vtype4,
            vtype5 ) braid_Int braid_##stype##StatusGet##param(braid_##stype##Status s, braid_↩
##vtype1 *v1, braid_##vtype2 *v2, braid_##vtype3 *v3, braid_##vtype4 *v4, braid_##vtype5 *v5);
```

11.3.2.8 ACCESSOR_HEADER_SET1

```
#define ACCESSOR_HEADER_SET1(
            stype,
            param,
            vtype1 ) braid_Int braid_##stype##StatusSet##param(braid_##stype##Status s, braid_↩
##vtype1 v1);
```

11.4 braid_test.h File Reference

Functions

- braid_Int braid_TestInitAccess (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestSpatialNorm (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnSCoarsen coarsen, braid_Pt↩ FcnSRefine refine)
- braid_Int braid_TestResidual (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real dt, braid↩ _PtFcnInit myinit, braid_PtFcnAccess myaccess, braid_PtFcnFree myfree, braid_PtFcnClone clone, braid_Pt↩ FcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnResidual residual, braid_PtFcnStep step)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_↩ Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_Pt↩ FcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnSCoarsen coarsen, braid_PtFcnSRefine refine, braid_PtFcnResidual residual, braid_↩ PtFcnStep step)

11.4.1 Detailed Description

Define headers for XBraid user-test routines.

This file contains headers for the user to test their XBraid wrapper routines one-by-one.

Index