Manual de Referencia de PyACTS: PyBLACS, PyPBLAS y PyScaLAPACK

Release 1.0

Vicente Galiano
Violeta Migallón
Jose Penadés
Tony Drummond

12 de diciembre de 2006

www.pyacts.org
Universidad Miguel Hernández de Elche, (Spain)
Universidad de Alicante, (Spain)
Lawrence Berkeley National Laboratory, (U.S.A.)
Email: support@pyacts.org

Resumen

Este manual documenta la utilización de las liberías incluidas en la colección PyACTS: PyBLACS, PyPBLAS y PyScaLAPACK. En este documento describiremos las funciones y utilidades desarrolladas para hacer de estas librerías una herramienta fácil y sencilla para aquellos científicos e investigadores que necesiten crear herramientas y calículos mediante un lenguaje sencillo y cómodo como Python pero que requieren de una alta capacidad computacional. PyACTS es un conjunto de librerías para Python que permiten importar las librerías BLACS, PBLAS y ScaLAPACK (pertenecientes a la colección ACTS) al entorno de programación Python.

ÍNDICE GENERAL

CAPÍTULO

ONE

Introducción

El presente documento forma parte del proyecto PyACTS. Este proyecto se centra en conseguir un interfaz automatizado y sencillo a las rutinas incluidas en las librerías ACTS desde el lenguaje interpretado Pyhton.

De este modo, investigadores ajenos a la metodología de la programación de altas prestaciones podrán escribir sus propios gracias a las facilidades incorporadas en el propio lenguaje Python y sobre todo en el paquete PyACTS que explicaremos en el presente documento.

CAPÍTULO

TWO

Instalación

2.1. Prerequisitos

Para la correcta instalación de PyScaLAPACK necesitamos tener correctamente instaladas las siguientes librerías y distribuciones en todas las estaciones del sistema multiproceso. Podremos distinguir dos grupos de librerías necesarias, el primero de ellos engloba el conjunto de librerías útiles en la computación de altas prestaciones. El segundo conjunto de librerías necesarias son módulos y extensiones del lenguaje interpretado Python.

- Librerías de Computación de Altas Prestaciones
  - `MPICH`
    Implementación del estándar MPI de código abierto. Disponible en http://www-unix.mcs.anl.gov/mpi/mpich/
  - `BLAS`
    Librería con rutinas de cálculo de operaciones básicas del álgebra lineal. Suele estar incluida en la mayor parte de las distribuciones actuales de `LINUX`. Una forma de averiguar si tenemos esta librería instalada en el sistema sería mediante el comando:

    ```
    locate libblas.a
    ```

    Pueden aparecer una o varias ubicaciones, deberemos elegir una de ellas para indicarlo en el archivo `setup.py` (ver sección **??**). En el caso que no se tuviera instalada la librería `BLAS`, podremos obtenerlas de la siguiente ubicación: http://www.netlib.org/blas/
  - `BLACS`
    Librerías de comunicaciones orientadas a datos de álgebra lineal. Estas librerias y la información necesaria para su instalación se encuentran disponibles en: http://www.netlib.org/blacs/
  - `ScaLAPACK`
    Librería que proporciona rutinas del álgebra lineal de Alto Rendimiento para sistemas de memoria distribuida mediante el intercambio de mensajes (MIMD). Esta librería es una continuación de LAPACK y en su distribución actual incluye rutinas útiles en el cálculo de matrices densas (`PBLAS`) así como funciones que permiten las resolución de sistemas lineales y obtención de valores propios.. Esta librería se encuentra disponible en la dirección: http://www.netlib.org/scalapack/
- Módulos y Extensiones de Python
  - `Python 2.1` (o superior)
    Python es un lenguaje interpretado que puede ahorrar un tiempo considerable durante el desarrollo de una aplicación. El interprete puede actuar de forma interactiva, esto facilita la comprobación inmediata de las nuevas ordenes insertadas. Además, los programas escritos en Python son muy compactos y legibles. Disponible en http://www.python.org/

- `Numeric Python (Numpy)`
  Conjunto de extensiones para Python que permite manipular eficientemente un elevado volúmen de datos organizado en matrices. Disponible en http://numpy.sourceforge.net/

2.2. Obtención de las fuentes

Para instalar la distribución PyScaLAPACK como parte del Proyecto PyACTS se ha de obtener la distribución de las fuentes de el sitio oficial cuya dirección es la siguiente :

http://www.pyacts.org

Una vez descargado el archivo, podemos descomprimirlo con el siguiente comando:

```
gunzip -d PyACTS_1.0.0.tgz | tar xfz -
```

Una vez descomprimido, se obtiene un conjunto de ficheros y directorios con la siguiente estructura:

```
PyACTS/
   /PyACTS_ScaLAPACK
      setup.py
      /SETUPS
         setup_LINUX.py
         setup_SP2.py
      /LIB
         /PyACTS/
            __init__.py
            PyScaLAPACK.py
            PyBLACS.py
            PyPBLAS.py
            PyScaLAPACK_Tools.py
         /SRC
            fortranobject.c
            pyscalapackmodule.c
            pyscalapackwrappers.f
         /EXAMPLES
            /exPyBLACS
            /exPyPBLAS
            /exPyScaLAPACK
            /exPyPNetCDF
            /test_pyscalapack
      /PyACTS_PETSc
         /EXAMPLES
         /SRC
         /LIB
      /PyACTS_SuperLU
         /EXAMPLES
         /SRC
         /LIB
```

La instalación del módulo PyScaLAPACK, PyPBLAS y PBLACS (como parte de las herramientas PyACTS) se realiza en un único paso mediante la instrucción:

```
python setup.py install
```

Se ha de tener en cuenta que esta instrucción instalará la distribución PyACTS en la ubicación de los paquetes de la plataforma donde se instale, para lo que tendrá que tener permisos de escritura en esa carpeta.

Si se desea generar la distribución sin instalarla en el sistema podremos ejecutar el siguiente comando:

```
python setup.py build
```

El proceso de compilación y construcción del paquete PyACTS utiliza la utilidad `distutils` incluida en las versiones superiores e igual a la 2.1. El instalador de este paquete únicamente deberá preocuparse de editar correctamente el archivo `setup.py` que se encuentra en la raiz de la carpeta `PyACTS_ScaLAPACK`.

En el siguiente punto trataremos en profundidad cada uno de los parámetros a establecer en este archivo de configuración.

2.3. Edición de setup.py

El proceso de instalación consta de una sola parte dependiente en su totalidad de la correcta configuración del archivo `setup.py`. En este archivo deberemos establecer la ubicación de los directorios de inclusión, y las librerías necesarias (comentadas en el apartado anterior) para el correcto enlazado y obtener así la librería compartida `_pycalapack.so`.

Por tanto, en el archivo `setup.py` deberemos modificar el valor de las ubicaciones de las librerías necesarias y el nombre de las mismas. Por defecto, en su distribución actual el valor de las mismas es el siguiente:

```
library_dirs_list=['./BUILD','/usr/local/BLACS/BLACS_PYTHON/LIB',
'/usr/local/mpich/lib','/usr/lib',
'/usr/local/SCALAPACK/SCALAPACK/'
]
libraries_list = [
    'scalapack',
    'blacsF77init_MPI-LINUX-0',
    'blacs_MPI-LINUX-0',
    'blacsF77init_MPI-LINUX-0',
    'blacsCinit_MPI-LINUX-0',
    'blacs_MPI-LINUX-0',
    'blacsCinit_MPI-LINUX-0',
    'blas',
    'mpich',
    'g2c']
```

Se ha de observar que `library_dirs_list` establece la ubicación o *path* donde el compilador buscará las librerías indicadas en `libraries_list`. Se ha de tener en cuenta, que el nombre del archivo que el compilador busca (mediante el parámetro `-lscalapack`) será `libscalapack.a`. Es decir, el nombre de los ficheros de librerías deberán comenzar por "`lib`" y terminar en "`.a`", donde este prefijo y sufijo no son indicados en la lista `libraries_list`.

En este punto se desea destacar que librerías como `blacsCinit_MPI-LINUX-0` aparecen repetidas veces en el listado. Esto se debe al propio proceso de enlazado y es necesario para una correcta construcción del objeto final.

Por otro lado en el proceso de compilación que realizará Python mediante `distutils` también deberemos indicar la ubicación de los archivos de inclusión y cabeceras referenciados en el código fuente de la distribución. La indicación

de las ubicaciones de los archivos de inclusión se realiza mediante el listado:

```
include_dirs_list = ['./SRC','/usr/local/mpich/include']
```

Una vez modificado el archivo con los valores adecuados se puede proceder a la ejecución de la construcción del paquete mediante el comando descrito en el apartado anterior.

```
python setup.py build
```

y si este proceso finaliza correctamente, un usuario con permisos de administrador podrá instalar el paquete en el sistema mediante:

```
python setup.py install
```

CAPÍTULO

THREE

MPIPython

Para poder ejecutar nuestras librerías de computación paralela necesitamos que nuestro interprete de Python sea capaz de ejecutar una aplicación bajo una plataforma de computación paralela bajo el interfaz de intercambio de mensajes MPI. Sin embargo, no es el caso , puesto que Python se caracteriza por ser un lenguaje de alto nivel que se ejecuta de modo mono-proceso en su distribución estándar. Por tanto, el interprete de Python incluido en la distribución estándar no se configura como una solución valida para poder ejecutar un mismo código en un sistema multiproceso.

Sin embargo, el interprete de comandos proporcionado en la distribución estándar de su página principal http://www.python.org/, se distribuye libremente bajo licencia de código abierto (GNU) y se encuentra disponible junto con una extensa librería estandar de programación para poder modificar esta distribución adaptando el interprete a nuestras necesidades.

Con el fin de obtener un interprete de Python que se pueda ejecutar en modo multiproceso, modificaremos el código fuente de la distribución estándar. De este modo necesitamos un nuevo interprete de Python que inicialice un entorno de computación paralela a través del paradigma de paso de mensajes. Ante esta necesidad tenemos dos soluciones posibles:

- `PyMPI`
  La extensión pyMPI está diseñada para proporcionar operaciones paralelas en Python en una arquitectura distribuida mediante MPI. Además de proporcionar una interfaz completa desde Python a las funciones avanzadas de MPI y a los comunicadores, pyMPI también incluye un interfaz simplificado que consigue que la programación sea más sencilla. Una de las formas más sencillas de utilizar pyMPI es de forma interactiva. Una vez inicializado pyMPI en la arquitectura correspondiente (de la forma normal en cada sistema con mpirun, prun, poe, etc. . .), se obtiene la identificación que Python está esperando comandos de entrada (>>>). En este momento, se esta ejecutando Python en modo multiprocesador tal y como se muestra a continuación.

  ```
  %mpirun -np 3 pyMPI
  >>> import mpi
  >>> print 'Soy el ', mpi.rank,' de ',mpi.size
  Soy el 0 de 3
  Soy el 2 de 3
  Soy el 1 de 3
  >>>
  ```

  Se puede comprobar en este ejemplo cómo se lanzan tres procesos en ejecución asíncrona puesto que cada se escribe su valor identificativo sin orden aparente. Los atributos `size` y `rank` del modulo importado `mpi` se corresponden con el número de procesos ejecutándose y con el identificador de cada uno de ellos respectivamente.

- `mpipython`
  Para facilitar el proceso de instalación y la utilización del paquete PyACTS, hemos desarrollado un interprete

de python en paralelo accesible en la dirección:

http://ideafix.umh.es/pyacts/mpipython.tgz

o bien a traves desde la página de inicio de la distribución oficial de PyACTS (http://ideafix.umh.es/pyacts/).

Se desea destacar que la utilización de cualquiera de las dos soluciones es totalmente válida y el paquete PyACTS ha sido probado con ambos interpretes y en múltiples plataformas (LINUX, IBM SP2,...). En el resto del presente capítulo describiremos el proceso de compilación y generación del ejecutable `mpipython` que será el interprete utilizado para ilustrar los ejemplos de cada una de las rutinas incluidas en el paquete PyACTS.

Por otro lado, queremos destacar que el código fuente del ejecutable `mpipython` no se adjunta en la actual distribución de PyACTS, sin embargo está accesible desde la página de la distribución antes descrita. Este intérprete se ha obtenido a partir de la distribución de *Scientific Python de Konrad Hinsen* . La colección de rutinas para Python de Konrad Hinsen proporciona un amplio conjunto de funciones y librerías muy útiles en la computación científica. El numero de rutinas y librerías es muy amplio y si se desea profundizar recomendamos la lectura de la referencia indicada. Sin embargo, para el problema que nos ocupa, sólo una parte del código nos resultaba útil en nuestro proposito de generar un interprete de Python multiproceso. De este modo, hemos indentificado las partes necesarias de esta librerías y las adjuntamos en la actual distribución.

3.1. Prerequisitos de mpipython

Para compilar y generar el interprete *mpipython* es necesario tener instalado en las plataforma correspondientes las siguientes librerías:

- `Python 2.1`
  disponible en http://www.python.org/
- `MPICH`
  Implementación del estándar MPI de código abierto, disponible en
  http://www-unix.mcs.anl.gov/mpi/mpich/
- `Numeric Python (Numpy)`
  Conjunto de extensiones para Python que permite manipular eficientemente un elevado volúmen de datos organizado en matrices. Disponible en
  http://numpy.sourceforge.net/

Se ha de tener en cuenta que debido a que el sistema es multiplataforma, estos requerimientos se establecen en todas aquellas estaciones o que pertenecen al sistema de computación en paralelo, por lo que se debe tener instalados los requerimientos en todas las máquinas donde fuere ejecutado `mpipython` y no sólo en aquella donde se compile.

3.2. Instalación

La distribución de mpipython se puede obtener en la dirección antes indicada:
http://ideafix.umh.es/pyatcs/mpipython.tgz

que una vez descargada podremos descomprimirla mediante el comando:

```
gunzip -c mpipython.tgz | tar xfz -
```

La estructura de archivos y directorios dentro de la carpeta `MPIPYTHON` es la siguiente:

```
MPIPYTHON\
         \src
         \bin
         \include
         \console
```

Cada una de las carpetas contiene un código con una funcionalidad determinada:

- `src`
  Contiene el código fuente con las funciones principales que generan mpipython.
- `include`: Archivos de inclusión necesarios extraídos de las librerías *Numeric Pyhton* y *Scientific Python de Konrad Hinsen*.

Para llevar a cabo la instalación de mpipython de forma independiente a la distribución del paquete PyACTS, nos situaremos en el directorio `MPIPYTHON` . Antes de lanzar el proceso de compilación deberemos editar el fichero `Makefile` y modificar los siguientes campos:

- `HOME`: Ubicación del propio fichero que ese está editando, esto es, directorio raíz de MPIPYTHON.
- `PYTHON_INCLUDE`: Ubicación del directorio `include` perteneciente a la distribución de Python instalada en el sistema.
- `PYTHON_CONFIG`: Ubicación del directorio `config` perteneciente a la distribución de Python instalada en el sistema.

El proceso de compilación hace uso del compilador mpicc incluido dentro de la distribución *MPICH*. Para lanzar el proceso de compilación deberemos ejecutar el siguiente comando:

```
[root@localhost MPIPYTHON]# make
```

El resultado es el ejecutable `mpipython` que se obtiene en el directorio `HOME` y que utilizaremos en el resto del documento como interprete de Python en paralelo.

3.3. Testeo

Una vez realizada la compilación podemos ejecutar algunos scripts (códigos) que se encuentran en el directorio raíz. En el directorio se encuentran algunas distribuciones para realizar pruebas de la correcta compilación. Antes de realizar las pruebas debemos recordar que el interprete puede ser ejecutado en dos modos : interprete e interactivo.

- *Modo interprete*: Cuando se ejecuta `python file.py` , el interprete lee el fichero y lo interpreta realizando las acciones que contenga. Para probar mpipython en este modo de funcionamiento podremos hacerlo del siguiente modo:

```
[vgaliano@localhost MPIPYTHON]$ mpirun -np 4 ./mpipython runme.py
hello world
hello world
hello world
hello world
[vgaliano@localhost MPIPYTHON]$
```

Analizaremos el comando introducido detallando el significado de cada una de las lineas:

- `mpirun`: Lanza el ejecutable que se indique a continuación en el número de procesos que indique el parámetro `-np`, en este ejemplo se lanza el interprete de Pyhton en cuatro procesadores.
- `mpipython`: Este es el ejecutable que hemos creado, es decir, el interprete de Python multiproceso. Este interprete se ejecutará en cada uno de los procesos y utilizara como fichero de entrada el fichero indicado a continuación.
- `runme.py`: Es el script por defecto que hemos utilizado en la distribución y realiza un sencillo `hello world`. El codigo del script es muy sencillo gracias a la fácil sintaxis de Python.

```
print "hello world"
```

- *Modo Interactivo*: Cuando ejecutamos el interprete de Python de la distribución standard mediante `python`, el resultado es una consola de entrada de comandos con el siguiente aspecto:

```
[root@localhost root]# python
Python 2.3 (#3, Aug 15 2003, 23:25:23)
[GCC 3.2 20020903 (Red Hat Linux 8.0 3.2-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Los signos >>> significan la espera a la introducción de un nuevo comando. Cada uno de los comandos que vayamos introduciendo se iran ejecutando de forma progresiva e interactiva:

```
>>> que_si = 1
>>> if que_si:
...     print "Pues va a ser que si"
...
Pues va a ser que si
```

Este modo de funcionamiento tipo *Consola* también podemos obtenerlo en un sistema multiproceso. Este lo podemos conseguir mediante la distribución PyMPI comentada al principio del apartado o bien mediante mpipython proporcionado en la pagina de la distribución de PyACTS.

Si deseamos utilizar `mpipython` tendremos que hacer uso del directorio `console` que aparece dentro de la carpeta `MPIPYTHON`. Sin embargo queremos destacar un detalle importante: para ejecutar mpipython en el modo interprete de la distribución anterior es suficiente con los requerimientos indicados previamente y con el codigo incluido en `mpipython.tgz`. Pero si deseamos ejecutar mpipython en modo consola necesitaremos hacer uso de las librerías de Scientific Python y por tanto tendremos que tenerlas instaladas en todas las estaciones de sistema. Para obtener Scientific Python podemos hacerlo desde la siguiente dirección:

`http://starship.python.net/ hinsen/ScientificPython/`

Una vez instalado Scientific Python, podremos ejecutar el modo consola del siguiente modo:

```
[vgaliano@localhost MPIPYTHON]$ mpirun -np 4 ./mpipython ./Console/Console.py
Python 2.2.1 (#1, Aug 30 2002, 12:15:30)
[GCC 3.2 20020822 (Red Hat Linux Rawhide 3.2-4)] on linux2
Type "copyright", "credits" or "license" for more information.
(Parallel console, 4 processors)
>>> a=1
>>> print a
-- Processor 0 ---------------------------------------
1
-- Processor 1 ---------------------------------------
1
-- Processor 2 ---------------------------------------
1
-- Processor 3 ---------------------------------------
1
>>>
```

Observamos en este caso como ejecutamos la consola en cuatro procesos, y la entrada del dato (`a=1`) se produce en todos ellos. Del mismo modo, cuando queremos mostrar en pantalla el valor de a que posee cada proceso , el nombre de la variable es el mismo pero identificamos a que proceso se corresponde mediante la identificación `Processor`.

CAPÍTULO

FOUR

Testeo del Paquete

Una vez finalizado el proceso de compilación e instalación, podemos comprobar si se ha instalado correctamente el paquete PyACTS. Para ello tendremos que ejecutar el *script* situado en la carpeta `EXAMPLES/PyACTS_test.py`. Podremos ejecutar este *script* en uno o varios procesadores en una arquitectura paralela mediante el siguiente comando.

```
mpirun -np 4 mpipython PyACTS_test.py
```

Se ha de tener en cuenta que `mpipython` se corresponde con el ejecutable del interprete de Python en paralelo creado en el capítulo 3.

El *script* `PyACTS_test.py` realiza llamadas a diversas funciones de los módulos PyBLACS, PyPBLAS y PyScaLAPACK y en el caso que finalice la ejecución correctamente se indicará al final del mismo. En la ejecución de este código de prueba se realizan las siguientes acciones:

- Importación del Paquete PyACTS con todos sus módulos PyBLACS, PyPBLAS y PyScaLAPACK.
- Inicialización de una malla de procesos automática
- Ejecución de rutinas incluidas en las PyPBLACS: Envío de un vector desde un proceso a el resto mediante comunicaciones punto a punto.
- Ejecución de rutinas incluidas en las PyPBLAS:
  - PyPBLAS Nivel 1: `pvaxpy`
  - PyPBLAS Nivel 2: `pvger`
  - PyPBLAS Nivel 3: `pvgemm`
- Ejecución de rutinas incluidas en las PyScaLAPACK:
  - Resolución de un sistema de ecuaciones mediante `pvgesv`
  - Cálculo de los valores únicos de una matriz mediante `pvgesvd`

El resultado de la ejecución de *script* de prueba mostraría el siguiente resultado:

```
************************************************
Testing PyACTS Distribution
************************************************
Number of processors: 4
I am : 0
Automatic Grid Configuration (nprow x npcol): 2 x 2
Block Size: 2 x 2
**********Testing PyBLACS Routines**********
[ 0 , 0 ] sends to [ 0 , 0 ]; a= [ [0 1 2 3 4 5 6 7]]
[ 0 , 0 ] sends to [ 0 , 1 ]; a= [ [0 1 2 3 4 5 6 7]]
[ 0 , 0 ] sends to [ 1 , 0 ]; a= [ [0 1 2 3 4 5 6 7]]
[ 0 , 0 ] sends to [ 1 , 1 ]; a= [ [0 1 2 3 4 5 6 7]]
**********Testing PyPBLAS Routines**********
-->Testing Level 1 (pvaxpy)
-->Testing Level 2 (pvger)
-->Testing Level 3 (pvgemm)
**********Testing PyScaLAPACK Routines**********
-->Testing pvgesv: Resolve Linear system
-->Testing pvgesvd: Resolving EigenValues
I am : 1
[ 0 , 1 ] receives a= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
I am : 2
[ 1 , 0 ] receives a= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
I am : 3
[ 1 , 1 ] receives a= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
```

CAPÍTULO

FIVE

Proyecto PyACTS

5.1. Introducción

Como ya hemos comentado en los capítulos precedentes, el nombre del paquete que vamos a distribuir (`PyACTS`) recibe el nombre de la Colección ACTS (*Advanced CompuTational Software*). Ésta es un conjunto de herramientas software muy útiles para los desarrolladores de programas en plataformas paralelas y difiere de otras herramientas para la computación paralela en el hecho que enfoca su funcionalidad en los niveles inferiores de una aplicación, proporcionando librerías que podrán ser utilizadas desde distintos lenguajes (C, Fortran 77/90, C++, etc.). La mayor parte de las herramientas han sido desarrolladas por Universidades y Laboratorios del Departamento de Energía de Estados Unidos [**?**].

La mayoría de las librerías incluidas en la colección ACTS están programadas en C (BLACS, Hypre, PETSc), otras en C++ (OPT++)e incluso en Fortran (ScaLAPACK). Estas librerías están diseñadas para su ejecución en procesadores paralelos utilizando el estándar MPI (ver sección **??**)para realizar la comunicación entre procesos.

La Colección ACTS no es un conjunto cerrado de herramientas y se permite la adicción de nuevas herramientas siempre que éstas cumplan con unos criterios mínimos para la Computación Científica ([**?**]). Las funcionalidades de las librerías pertenecientes a la Colección ACTS se pueden clasificar en cuatro grupos:

- *Numéricas*:
  Implementan métodos numéricos de resolución de sistemas densos y dispersos, en este grupo incluimos: Aztec ([**?**],[**?**]), Hipre [**?**], OPT++ ([**?**],[**?**]), PETSc [**?**], ScaLAPACK [**?**], SUNDIALS [**?**], SuperLU [**?**] y TAO ([**?**], [**?**]).
- *Entornos de trabajo*:
  Proporcionan la infraestructura necesaria que permite manejar la complejidad de la programación paralela (distribución de matrices, comunicación de información del entorno, etc.) pero no implementan métodos numéricos. Este grupo de herramientas está formado por Global Arrays [**?**], y Overture [**?**].
- *Soporte para la ejecución de aplicaciones*:
  Agrupa utilidades en el nivel de aplicación como puede ser el análisis del rendimiento y la visualización remota. Librerías pertenecientes a este bloque son CUMULUS ([**?**], [**?**]), GLOBUS ([**?**], [**?**]) y TAO [**?**].
- *Soporte para el desarrollador* :
  Proporcionan la infraestructura y las herramientas que ayudan en el desarrollo de una aplicación de este tipo y mejoran el rendimiento de estas. Una librería perteneciente a este tipo es la librería ATLAS [**?**].

El presente documento, conforma el primer paso para conseguir un entorno de programación integrado que permita desde el interfaz Python, poder hacer uso de las distintas librerías que forma la colección ACTS interactuando entre ellas de una forma sencilla para el usuario final pero sin perjuicio del rendimiento ni la capacidad de este tipo de librerías para la computación de altas prestaciones.

En capítulos posteriores describiremos los módulos PyBLACS, PyPBLAS y PyScaLAPACK que ofrecen un interfaz automatizado a las librerías BLACS, PBLAS y ScaLAPACK respectivamente.

5.2. El Objeto PyACTS

Para facilitar la interrelación entre las diferentes librerías que componen la colección ACTS, se ha definido un nuevo objeto denominado PyACTS. El objetivo principal de este tipo de objetos es servir como contenedor de los datos, y descriptores utilizados por las librerías de PyACTS.

Este objeto tiene tres propiedades:

- `ACTS_lib`: Indicador del formato del descriptor `desc` y del archivo de datos `data` a partir de la librería de la colección ACTS que queremos hacer uso. Los valores identificativos de cada uno de los formatos de las librerías son:
  - `ACTS_lib=1`:Distribución cíclica 2-D utilizada en los módulos:
    - PyPBLAS
    - PyScaLAPACK
- `desc`: Descriptor de la matriz distribuida. En este descriptor se almacena generalmente un vector cuyo formato y significado depende de la librería ACTS que deseemos utilizar indicado en `ACTS_lib`.
- `data`: Contiene los datos correspondientes a la distribución cíclica 2D que le corresponde a cada proceso en función de la configuración de la malla de procesos (`nprow` x `npcol`), el tamaño de bloque (`mb` x `nb`) y el identificador de cada proceso dentro de la malla (`myrow` x `mycol`).

De este modo, las rutinas incluidas en los módulos referidos esperan que los tipos de datos de entrada sean los correctos en cada caso, y en muchos de ellos esperan que las matrices sean instancias `PyACTS`. Para convertir, leer desde diferentes formatos a PyACTS y viceversa se incluyen diversas utilidades que serán explicadas en el siguiente capítulo.

CAPÍTULO

SIX

Herramientas: Herramientas PyACTS

Dentro del paquete PyACTS, utilizaremos un conjunto de rutinas que nos serán útiles para inicializar, consultar y liberar los recursos necesarios para poder ejecutar las rutinas incluidas en el paquete tanto en PyBLACS, PyPBLAS y PyScaLAPACK.

Este conjunto de rutinas se pueden agrupar en tres funcionalidades básicas:

- Inicialización y Liberación de recursos.
- Verificación de variables PyACTS.
- Consulta y Generación de variables PyACTS.
- Conversión de formatos PyACTS.

Veremos las rutinas de cada uno de estos grupos mostrando ejemplos que ilustren su finalidad y comportamiento:

6.1. Inicialización y Liberación

Estas rutinas han de ser ejecutadas para inicializar o liberar recursos y deberán estar presentes de forma anterior y posterior (respectivamente) a la ejecución de las rutinas de los módulos PyBLACS, PyPBLAS y PyScaLAPACK.

6.1.1. gridinit

`gridinit([file_config=",mb,nb,nprow,npcol])`

Esta rutina se encarga de inicializar la malla de procesos en función de los parámetros que se le pasan y en ausencia de estos opta por configurar una malla lo mas cuadrada posible.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `file_config`: Fichero de configuración de la matriz a inicializar. La configuración de la malla se establece en función de los valores almacenados en el fichero.
  - `nb=0`: Número de filas de la matriz bloque en la distribución cíclica 2D a realizar. En el caso que no se indique este parámetro, se establece un valor por defecto de 64.
  - `mb=0`: Número de columnas de la matriz bloque en la distribución cíclica 2D a realizar. En el caso que no se indique este parámetro, se establece un valor por defecto de 64.

- `nprow=0`: Número de filas de la malla de procesos para realizar la distribución cíclica 2D. En el caso que no se indique este parámetro, se establece un valor por defecto de buscando una malla de procesos lo más cuadrada posible e intentando utilizar todos los procesos.
- `ncol=0`: Número de filas de la malla de procesos para realizar la distribución cíclica 2D. En el caso que no se indique este parámetro, se establece un valor por defecto de buscando una malla de procesos lo más cuadrada posible e intentando utilizar todos los procesos.

- `Parametros de Salida (no tiene)`

A pesar de no tener parámetros de salida, esta función establece los valores de las siguientes variables que serán utilizadas en cada proceso por el conjunto de rutinas incluidas en este paquete:

- `PyACTS.mb`
- `PyACTS.nb`
- `PyACTS.nprow`
- `PyACTS.npcol`
- `PyACTS.myrow`
- `PyACTS.mycol`
- `PyACTS.ictxt`

6.1.2. gridexit

```
gridexit()
```

Esta rutina se encarga de liberar los recursos y la malla de procesos una vez realizados los cálculos. Esta rutina es muy sencilla y no posee ningún parámetro de entrada ni de salida. Se ha de tener en cuenta que una vez liberados los recursos no podremos realizar nuevas llamadas a las rutinas de PyPBLAS o PyScaLAPACK a no ser que volvamos a inicializar una malla mediante `gridinit`.

6.2. Verificación de variables PyACTS

6.2.1. IsACTSlib

```
IsACTSlib(ACTS_var)
```

Devuelve mediante una expresión booleana que indica si la variable pasada como parámetro `ACTS_var` tiene el formato PyACTS válido.

6.3. Consulta y Generación de variables PyACTS

6.3.1. readACTSdesc

```
readACTSdesc(ACTSArray)
```

Devuelve el descriptor de la variable pasada como parámetro `ACTSArray`.

6.3.2. ACTSgetdesc

`desc=ACTSgetdesc(a[,ACTS_lib,ia,ja])`
Esta rutina devuelve el descriptor `desc` de la matriz global `a` de tipo `Numeric` pasada como parámetro para la librería indicada por el valor `ACTS_lib`. Este descriptor podrá ser utilizado para construir el Array ACTS o bien para crear una nueva matriz distribuida a partir de su descriptor. Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `a`: Matriz global de tipo `Numeric`. Al indicar que es una matriz global queremos indicar que el proceso almacena todos los valores de la matriz por tanto es una matriz que aún no ha sido distribuida. Esto implicaría una limitación en la escalabilidad de este código.
  - `ACTS_lib`: Por defecto, `ACTS_lib`=1 y por tanto se creará un descriptor acorde con las librerías PBLAS y ScaLAPACK utilizando una distribución cíclica 2D.
  - `ia`: Puntero dentro de la memoria local que indicaría la primera fila de la matriz local que será utilizada en el cálculo de la rutina. Por defecto, el valor utilizado internamente es 1.
  - `ja`: Puntero dentro de la memoria local que indicaría la primera columna de la matriz local que será utilizada en el cálculo de la rutina. Por defecto, el valor utilizado internamente es 1.
- `Parámetros de Salida`
  - `desc`: Matriz de 9 elementos con el formato adecuado para ser utilizado como descriptor de la matriz distribuida de a.

6.3.3. ACTSmakedesc

`desc=ACTSmakedesc(m,n[,ACTS_lib,ia,ja])`
Mientras que la rutina `ACTSgetdesc` creaba un descriptor a partir de una matriz dada (de tipo `Numeric`), la rutina `ACTSmakedesc` obtendrá un descriptor a partir del tamaño de la matriz indicado mediante los parámetros de entrada `m` y `n`. Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `m`: Numero de filas de la matriz global de tipo `Numeric`.
  - `n`: Numero de filas de la matriz global de tipo `Numeric`.
  - `ACTS_lib`: Por defecto, `ACTS_lib`=1 y por tanto se creará un descriptor acorde con las librerías PBLAS y ScaLAPACK utilizando una distribución cíclica 2D.
  - `ia`: Puntero dentro de la memoria local que indicaría la primera fila de la matriz local que será utilizada en el cálculo de la rutina. Por defecto, el valor utilizado internamente es 1.
  - `ja`: Puntero dentro de la memoria local que indicaría la primera columna de la matriz local que será utilizada en el cálculo de la rutina. Por defecto, el valor utilizado internamente es 1.
- `Parámetros de Salida`
  - `desc`: Matriz de 9 elementos con el formato adecuado para ser utilizado como descriptor de la matriz distribuida de a.

Se ha de tener en cuenta que estas dos ultimas rutinas son muy similares las siguientes llamadas devolverían el mismo resultado:

```
desc=ACTSgetdesc(a)
desc=ACTSmakedesc(a.shape)
```

6.3.4. getdims

```
m,n=getdims(x)
```

La rutina `getdims` obtiene las dimensiones de una matriz que reside en un único proceso (aquel con `PyACTS.iread==1`) en todos los procesos. Esta rutina proporciona una forma sencilla de poder conocer el valor de una determinada matriz que todavia no ha sido distribuida ni creado su descriptor.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `x`: El proceso con `PyACTS.iread==1` contendrá la matriz de tipo `Numeric`, mientras que el resto pueden utilizar cualquier valor, por ejemplo `None`.
- `Parámetros de Salida`
  - `m`: Numero de filas de la matriz global de tipo `Numeric`.
  - `n`: Numero de filas de la matriz global de tipo `Numeric`.

De este modo, todos los procesos pueden obtener de una forma sencilla el tamaño de la matriz a distribuir:

```
#Send and receive the dims of the array
m,n=getdims(a)
```

6.3.5. readconfig

```
mb,nb,nprow,npcol=readconfig(iam,nprocs,file_config,mb0,nb0,nprow0,npcol0)
```

La rutina `readconfig` obtiene la configuración de la malla de procesos desde un archivo de configuración que indicaremos mediante `file_config` o bien la calcula a partir del número de procesos disponbibles. En el caso que la calcule de forma automática, se buscará una matriz de procesos lo más cuadrada posible y que agrupe el mayor número posible de procesos con los que se ha iniciado la ejecución (`mpirun -np N mpipython example.py`, donde `N` es el número de procesos). Por ejemplo, si `N=4` entonces `nprowxnpcol=2x2`, o si si `N=7` se obtendría `nprowxnpcol=3x2`.

Por otro lado, con respecto a la malla de procesos se puede indicar cualquiera de los parámetros (`nprow` o `npcol`)y se calcularán el resto en función del indicado y buscando utilizar el mayor número de procesos posible. Por ejemplo, si `N=9` e indicamos `nprow=4`, en lugar de obtener una configuración por defecto `nprowxnpcol=3x3`, obtendríamos `nprowxnpcol=4x2`. De este modo, podremos seleccionar la configuración de la malla de procesos que deseemos con determinados fines, como pudiera ser el de ajustar la malla de procesos a las formas de las matrices de datos. Siguiendo esta línea, si deseamos multiplicar dos vectores, sería conveniente utilizar una configuración `Nx1`.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `iam`: Indicador del proceso.
  - `nprocs`: Indicador del número de procesos con los que se ha inciado la aplicación.
  - `file_config` (opcional): Fichero de configuración donde leer esta configuración.
  - `mb` (opcional): Número de filas de la matriz bloque para la distribución cíclica 2D.
  - `nb` (opcional): Número de columnas de la matriz bloque para la distribución cíclica 2D.

- Parámetros de Salida
  - nprow: Numero de filas de la matriz de procesos.
  - npcol: Numero de columnas de la matriz de procesos.

6.4. Conversión de variables PyACTS

6.4.1. Num2PyACTS

a_acts=Num2PyACTS(a,ACTS_lib)
Esta rutina será muy utilizada en la mayor parte de los ejemplos de los capítulos posteriores en los que introduciremos ejemplos de PyBLACS, PyPBLAS y PyScaLAPACK. La rutina Num2PyACTS devuelve una variable de tipo ACTS Array a partir de una variable de tipo Numeric y del indicador de la librerías ACTS que deseamos aplicar.

Se ha de tener en cuenta que esta función de forma interna realiza las siguientes acciones:

1. A partir de la matriz, calcula y distribuye el tamaño de la matriz
2. Cada proceso crea las matrices locales para recibir el dato
3. Se crea el descriptor en cada proceso
4. Se distribuyen los datos de acuerdo a ese descriptor
5. Se crea la variable ACTSArray que contiene las tres propiedades: ACTS_lib, data y desc.

Las características de cada uno de los parámetros de entrada y salida son:

- Parámetros de Entrada
  - a: Matriz de tipo Numeric a distribuir.
  - ACTS_lib: Indicador de la librería de las ACTS utilizada. Por defecto, ACTS_lib=1 y por tanto se creará un descriptor acorde con las librerías PBLAS y ScaLAPACK utilizando una distribución cíclica 2D.
- Parámetros de Salida
  - a_acts: Matriz de tipo PyACTS con las propiedades ACTS_lib, data y desc "preparada" para ser utilizada como variable en cualquiera de las funciones del paquete PyACTS que requiera una variable de este tipo como dato de entrada.

6.4.2. PyACTS2Num

a=PyACTS2Num(a_acts)
Esta rutina se puede considerar la inversa de la rutina Num2PyACTS y también será utilizada en la mayor parte de los ejemplos de los capítulos posteriores. La rutina PyACTS2Num devuelve una variable de tipo Numeric a partir de una variable de tipo PyACTS.

Se ha de tener en cuenta que esta función de forma interna realiza las siguientes acciones:

1. A partir de la variable PyACTS y por tanto de su descriptor, se crea en el proceso con PyACTS.iread=1 una matriz de las dimensiones de la matriz total.
2. Todos los procesos envían al proceso PyACTS.iread=1 los datos y éste los copia en sus índices correspondientes.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `a_acts`: Variable de tipo `PyACTS` donde sus datos se encuentran distribuidos en los procesos que componen la malla con una distribución cíclica 2D.
- `Parámetros de Salida`
  - `a`: Matriz de tipo `Numeric` cuyos datos residen en un único proceso (`PyACTS.iread=1`).

6.4.3. Rand2PyACTS

```
a_acts=Rand2PyACTS(m,n,ACTS_lib)
```

Esta rutina crea una variable de tipo PyACTS donde los datos creados son aleatorios y el tamaño global de la misma es $m \times n$. Se ha de tener en cuenta, que en esta rutina la creación de las datos aleatorios se realizan en cada proceso por lo que no se realiza distribución de datos. La única distribución que se realiza es el tamaño global de la matriz a crear $m \times n$. A partir de este tamaño, cada proceso calculará el tamaño de su matriz local que dependerá de su configuración dentro de la malla de procesos.

Se ha de tener en cuenta que esta función de forma interna realiza las siguientes acciones:

1. Se crea el descriptor en cada proceso
2. Distribuye el tamaño de la matriz a crear
3. Cada proceso crea las matrices locales aleatorias de las dimensiones calculadas en función de la configuración de la malla de procesos.
4. Se crea la variable ACTSArray que contiene las tres propiedades: `ACTS_lib`, `data` y `desc`.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `m`: Numero de filas de la matriz global.
  - `n`: Numero de filas de la matriz global.
  - `ACTS_lib`: Indicador de la librería de las ACTS utilizada. Por defecto, `ACTS_lib`=1 y por tanto se creará un descriptor acorde con las librerías PBLAS y ScaLAPACK utilizando una distribución cíclica 2D.
- `Parámetros de Salida`
  - `a_acts`: Matriz de tipo PyACTS con las propiedades `ACTS_lib`, `data` y `desc` "preparada" para ser utilizada como variable en cualquiera de las funciones del paquete PyACTS que requiera una variable de este tipo como dato de entrada.

```
alpha_acts=Scal2PyACTS(alpha,ACTS_lib)
```

Esta rutina crea una variable de tipo PyACTS donde el dato (ya que es un escalar) reside en la propiedad `data`. La utilidad de está función se centra en automatizar la distribución del escalar en todos los procesos que pertenecen a la configuración de la malla de procesos.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`

- `alpha`: Número escalar.
- `ACTS_lib`: Indicador de la librería de las ACTS utilizada.

- `Parámetros de Salida`
  - `alpha_acts`: Escalar de tipo PyACTS con las propiedades `ACTS_lib`, `data` y `desc` "preparado" para ser utilizada como variable en cualquiera de las funciones del paquete PyACTS que requiera una variable de este tipo como dato de entrada.

Ejemplos en la utilización de esta rutina se pueden encontrar en los capítulos posteriores PyBLACS, PyPBLAS y PyScaLAPACK.

6.4.4. PNetCDF2PyACTS

`a_acts=PNetCDF2PyACTS(a_pnetcdf,ACTS_lib)`
Para poder hacer uso de esta rutina es necesario tener instalada el paquete PyPNetCDF distribuido en :
http://ideafix.umh.es/pypnetcdf

Una vez instalado este paquete correctamente podremos hacer que interactúe con el paquete PyACTS mediante esta y otras rutinas que explicaremos a continuación. La principal finalidad de esta rutina es la de obtener los datos desde una instancia de un objeto PyPNetCDF para conseguir una variable de tipo PyACTS que nos permita utilizarla en las rutinas PyBLACS, PyPBLAS y PyScaLAPACK.

La utilización de esta rutina trata de ser muy sencilla, sin embargo para poder comprender su funcionamiento será necesario también entender el funcionamiento básico del paquete PyPNetCDF.

El proceso que se sigue en esta rutina será el siguiente:

1. Cada proceso lee los parametros de interes de la instancia PyPNetCDF para crear el descriptor.
2. Calcula el tamaño de sus matrices locales conforme a la distribución cíclica 2D, a la configuración de la malla de procesos y al tamaño de la matriz global, obtenido de las dimensiones de la variable PNetCDF.
3. Cada proceso lee de forma independiente los datos situados en los índices correspondientes en función de la distribución cíclica 2D.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `a_pnetcdf`: instancia de un objeto de tipo `PyPNetCDF.Variable`.
  - `ACTS_lib`: Indicador de la librería de las ACTS a la que queremos convertir.
- `Parámetros de Salida`
  - `a_acts`: Matriz de tipo PyACTS con las propiedades `ACTS_lib`, `data` y `desc` "preparada" para ser utilizada como variable en cualquiera de las funciones del paquete PyACTS que requiera una variable de este tipo como dato de entrada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
import PyACTS.pyscalapack as pyscalapack
from PyPNetCDF.PNetCDF import *
from RandomArray import *
from Numeric import *
#Initiliaze the Grid
PyACTS.gridinit()
file = PNetCDFFile('data_pvgemm.nc', 'r')
if PyACTS.iread==1:
  print "Example Using PyPBLAS 3 (PvGEMM) y and PyPNetCDF "
  print "nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
  print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb

a = file.variables['a']
b = file.variables['b']
c = file.variables['c']
#We convert NetCDF Variables to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha1=PNetCDF2ScalPyACTS(file.alpha[0],ACTS_lib)
beta1=PNetCDF2ScalPyACTS(file.beta[0],ACTS_lib)
a1=PNetCDF2PyACTS(a,ACTS_lib)
b1=PNetCDF2PyACTS(b,ACTS_lib)
c1=PNetCDF2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c1= PyPBLAS.pvgemm(alpha1,a1,b1,beta1,c1)
#Create Results File in NetCDF Format
file2 = PNetCDFFile('result_data_pvgemm.nc', 'w')
file2.title = "Result of operation with pvgemm"
file2.version = 1
for dim in file.dimensions:
  file2.createDimension(dim, file.dimensions[dim])
c_result = file2.createVariable('c_result', c.vartypecode, c.dimensions)
file2.enddef()
#Write results from PyACTS data to a  NetCDF File
c_result=PyACTS2PNetCDF(c1,c_result)
file2.close()
file.close()
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvgemm.py
```

6.4.5. PyACTS2PNetCDF

`a_netcdf=PyACTS2PNetCDF(a_acts,a_netcdf)`
Para poder hacer uso de esta rutina es necesario tener instalada el paquete PyPNetCDF distribuido en : http://ideafix.umh.es/pypnetcdf

La principal finalidad de esta rutina es la de almacenar los datos desde una instancia de un objeto PyACTS Array a una instancia de tipo PyPNetCDF, para posteriormente guardar los datos. La utilización de esta rutina trata de ser muy

sencilla, sin embargo para poder comprender su funcionamiento será necesario también entender el funcionamiento básico del paquete PyPNetCDF.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parámetros de Entrada`
  - `a_acts`: instancia de un objeto de tipo `PyACTS` que contiene los datos distribuidos para ser almacenados en una instancia `PyPNetCDF.Variable`.
  - `a_pnetcdf`: instancia de un objeto de tipo `PyPNetCDF.Variable` donde se almacenará y manejarán los datos.
- `Parámetros de Salida`
  - `a_pnetcdf`: instancia de un objeto de tipo `PyPNetCDF.Variable` con los datos almacenados

En la rutina anterior, se ha adjuntado un ejemplo que también ilustra la utilización de PNetCDF2PyACTS para guardar los cálculos obtenidos.

CAPÍTULO

SEVEN

PyBLACS

En el presente capítulo describiremos los principales interfaces que se han desarrollado para simplificar las funciones de acceso a las librerías `BLACS`. Las rutinas incluidas dentro de las librerías `BLACS` proporcionan un interfaz de comunicación para utilizar en funciones básicas del Álgebra Lineal.

Los interfaces creados en estas librerías se diferencian en dos grupos:

- `Sencillos`: El acceso a las rutinas se realiza mediante un interfaz automatizado, donde muchos de sus parámetros e inicializaciones se realizan de forma automática. De este modo, el usuario deberá únicamente preocuparse de los datos a enviar.
- '`Avanzados`': Se accede de forma directa a los interfaces de las librerías de las BLACS. Todos los parámetros que se han de especificar en las rutinas originales, se deberán indicar tambien en este interfaz de `Python`.

Realizaremos la siguiente clasificación del conjunto de rutinas a las que vamos a tener acceso mediante el interfaz sencillo:

- **Inicialización**
  - 7.2.1pinfo
  - 7.2.2setup
  - 7.2.3get
  - 7.2.4gridinit
  - 7.2.5gridmap
- **Destrucción**
  - 7.3.4freebuff
  - 7.3.2gridexit
  - 7.3.1abort
  - 7.3.3exit
- **Informativas**
  - 7.4.1gridinfo
  - 7.4.3pnum
  - 7.4.2pcoord
- **Envío**
  - 7.7.1gesd2d

    - 7.5.2trsd2d
- **Recepción**
    - 7.6.1gerv2d
    - 7.6.2trrv2d
- **Difusión**
    - **??**gebs2d
    - 7.7.2trbs2d
- **Recolección**
    - 7.8.1gebr2d
    - 7.8.2trbr2d
- **Operaciones Combinadas**
    - 7.9.1gsum2d
    - 7.9.2gamx2d
    - 7.9.3gamn2d
- **Varios**
    - **??**barrier

De este modo, describiremos sección a sección en este capítulo cada uno de los grupos funcionales de las rutinas incluidas dentro de las PyBLACS. Describiremos cual es su interfaz y mostraremos ejemplos de cada una de ellas.

*I*NICIALIZACIÓN	
`iam,nprocs=blacspinfo()`	Obtiene identidad y número de procesos (MPI)
`iam,nprocs = setup(nprocs)`	Obtiene identidad y número de procesos (PVM)
`val = get(what,[ictxt])`	Obtiene valores internos de las BLACS
`ictxt = gridinit(nprow,npcol[,order])`	Inicializa la malla con tamaño indicado
`ictxt = gridmap(ictxt,usermap,ldumap,nprow,npcol)`	Establece cada proceso dentro de una malla
*D*ESTRUCCIÓN	
`abort([ictxt,errornum])`	Mata todos los procesos BLACS cuando ha habido algun error
`gridexit(ictxt)`	Libera los recursos del contexto `ictxt`
`exit([ictxt])`	Libera TODOS los recursos de todos los contextos
`freebuff([ictxt,wait])` Libera el buffer de las BLACS	
*I*NFORMATIVAS	
`nprow,npcol,myrow,mycol=gridinfo([ictxt])`	Proporciona información de la malla con contexto `ictxt`
`prow,pcol = pcoord(pnum[,ictxt])`	Devuelve las coordenadas de un identificador dentro de la malla `ictxt`
`nprow,npcol,myrow,mycol=gridinfo([ictxt])`	Proporciona información de la malla con contexto `ictxt`
`num = pnum(prow,pcol,[ictxt])`	Proporciona el identificador a partir de las coordenadas de un proceso de la malla con contexto `ictxt`
*E*NVÍO	
`gesd2d(a,rdest,cdest,[ictxt,llda])`	Envia los datos de la matriz `a` hacia (`rdesc`,`cdest`)
`trsd2d(a,rdest,cdest,[ictxt,llda])`	Envia los datos de la matriz triangular `a` hacia (`rdesc`,`cdest`)
*R*ECEPCIÓN	
`a=gerv2d(a,rsrc,csrc,[ictxt,llda])`	Recibe los datos de la matriz `a` desde (`rsrc`,`csrc`)
`a=trrv2d(a,rsrc,csrc,[ictxt,llda,diag,llda])`	
Recibe los datos de la matriz triangular `a` desde (`rsrc`,`csrc`)	
*D*IFUSIÓN	
`gebs2d(a,[ictxt,scope,top,llda])`	Inicia la difusión al grupo de destinatarios indicados por `scope`
`trbs2d(a[,ictxt,scope,top,uplo,diag,llda])`	Inicia la difusión de una matriz triangular al grupo de destinatarios indicados por `scope`
`a=gebr2d(a,irsrc,icsrc[,ictxt,scope,top,lda])`	Recibe los datos provenientes de una difusión desde (`irsrc`,`icsrc`)
`a=trbr2d(a,irsrc,icsrc[,ictxt,uplo,diag,scope,top,lda])`	Recibe los datos provenientes de una difusión de una matriz triangular desde (`irsrc`,`icsrc`)
*O*PERACIONES COMBINADAS	
`a=gsum2d(a,rdest,cdest,[ictxt,scope,top,lda])`	Devuelve la suma de los elementos de la matriz
`a,ra,ca=gamx2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`	Devuelve el valor máximo del elemento y su posición en los datos
`a,ra,ca=gamn2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`	Devuelve el valor mínimo del elemento y su posición en los datos

7.2. Inicialización

En este capítulo nos centraremos en las principales rutinas de inicialización. Este tipo de tareas se realizan de forma automatica en algunas de las rutinas vistas en el capítulo 9 de PyScaLAPACK. Sin embargo, creemos conveniente que el usuario tenga acceso a estas rutinas por si deseara realizar sus propios calculos y necesita de algun tipo de herramienta para intercambiar datos relacionados con el álgebra lineal entre diferentes procesos en una computación en paralelo.

7.2.1. pinfo

`iam,nprocs=pinfo()`

La rutina '`pinfo`' proporciona información relativa al número de procesos y la identificación de cada uno de los procesos dentro de un entorno de computación paralela donde múltiples procesos intervienen. Esta rutina se usa para obtener informacion inicial del sistema antes de inicializar las `BLACS`. En todas las plataformas (excepto PVM), '`nprocs`' representa el numero de procesos disponibles para su uso (`nprows * npcols<= nprocs`). Las características de cada uno de

los parámetros de salida son:

- `Parametros de Entrada (No tiene)`
- `Parametros de Salida`
  - '`iam`': Número entero entre 0 y (nprocs -1) que identifica de forma unívoca a cada proceso.
  - '`nprocs`': Indica el número de procesos disponibles para el uso de las BLACS.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
print iam,nprocs
```

El resultado de este código es el siguiente:

```
1 4
2 4
3 4
0 4
```

7.2.2. setup

`iam,nprocs = setup(nprocs)`

Esta rutina únicamente tiene significado en el caso de esta trabajando bajo un entorno PVM. En otras plataformas, esta funcionalidad la proporciona la rutina `pinfo` (7.2.1). Las BLACS (y por tanto PyBLACS) asumen un sistema estático: el sistema se inicializa con un número dado de procesos, PVM proporciona un sistema dinámico permitiendo que los procesos se añadan y eliminen al sistema en ejecución. Las características de cada uno de los parámetros de salida son:

- `Parametros de Entrada (No tiene)`

- 'nprocs': Indica el número de procesos disponibles para el uso de las BLACS.
- Parametros de Salida
  - 'iam': Número entero entre 0 y (nprocs -1) que identifica de forma unívoca a cada proceso.
  - 'nprocs': Indica el número de procesos disponibles para el uso de las BLACS.

7.2.3. get

```
val = get(what,[ictxt])
```

La rutina get obtiene valores internos de las BLACS. Algunos valores se refieren a un identificador de contexto determinado 'idtxt'.

Las características de cada uno de los parámetros de salida son:

- Parametros de Entrada
  - 'what': Entero mediante el cual se solicita un determinado valor de las BLACS devuelto mediante la variable val. Los valores posibles son:
    - 'what=0': El contexto por defecto del sistema
    - 'what=1': El rango Id de las BLACS
    - 'what=2': El nivel de depuración con el que las librerías BLACS han sido compiladas.
    - 'what=10': El valor utilizado para definir el contexto de las BLACS
    - 'what=11': Número anillos en la topología multianillos utilizada
    - 'what=11': Número de ramificaciones en la topología en árbol utilizada.
- Parametros de Salida
  - 'val': Valor obtenido para el parámetro solicitado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
if ictxt <>-1:
        if iam==0:
                for i in [0,1,2,10,11,12]:
                        val = PyBLACS.get(i)
                        print "Proccess:",iam,"value[",i,"]=",val

PyBLACS.gridexit(ictxt)
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsget.py
Proccess: 0 Value[ 0 ]= 91
Proccess: 0 Value[ 1 ]= 0
Proccess: 0 Value[ 2 ]= 0
Proccess: 0 Value[ 10 ]= 138
Proccess: 0 Value[ 11 ]= 1
Proccess: 0 Value[ 12 ]= 1
```

7.2.4. gridinit

`ictxt = gridinit(nprow,npcol[,order])`

Mediante la rutina `gridinit` indicaremos cuantos procesos estarán contenidos dentro de la malla que deseamos establecer. Todas las rutinas de las `BLACS` deberán llamar a esta rutina o a su rutina homóloga (`gridmap`). Estas rutinas establecen cada proceso dentro de una malla. Cada malla `BLACS` está definida en un contexto (que representa su propio universo de paso de mensajes) , de este modo no se interfieren distintos contextos o distintas configuraciones de mallas. Esta rutina puede ser utilizada en repetidas ocasiones para definir contextos/mallas adicionales.

Esta rutina de creación de una malla establece variables internas a las BLACS que no deberán ser utilizadas antes de la llamada a la funcion `gridinit`.

Esta rutina crea una malla de procesos de tamaño `nprow x npcol`, y asigna a cada proceso un orden dentro de esa malla dependiendo del parámetro `order`.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - '`nprow`': Indica cuantas filas se han de configurar en la malla de procesos.
  - '`npcol`': Indica cuantas columnas se han de configurar en la malla de procesos.
  - '`order`':Indica cómo distribuir los procesos dentro de la malla BLACS. Las posibilidades son:
    - `order='R'`: (por defecto)Utiliza una ordenación de menor a mayor por filas.
    - `order='C'`: Utiliza una ordenación de menor a mayor por columnas.
    - `order=(otro valor)`: Utiliza una ordenación de menor a mayor por filas.
- `Parametros de Salida`
  - '`ictxt`': Entero de salida que indica el contexto BLACS que se ha creado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
if ictxt<>-1:
        print "Soy ",iam,".I'm in the grid. ictxt=",ictxt
else:
        print "Soy ",iam,".I'm not in the grid. ictxt=",ictxt
PyBLACS.gridexit(ictxt)
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 5 mpipython exPyblacsgridinit.py
Soy  4 .I'm not in the grid. ictxt= -1
Soy  3 .I'm in the grid. ictxt= 0
Soy  1 .I'm in the grid. ictxt= 0
Soy  2 .I'm in the grid. ictxt= 0
Soy  0 .I'm in the grid. ictxt= 0
```

Hemos de observar que el proceso n 4 (`iam=5`) no pertenece a la malla 2x2 creada. De este modo, únicamente aquellos procesos con `ictxt=0`, pertenecen a la malla.

7.2.5. gridmap

```
ictxt = gridmap(ictxt,usermap,ldumap,nprow,npcol)
```

Todos los programas que utilicen PyBLACS deberán llamar a esta rutina de inicialización o bien a una rutina similar como es `gridinit`. Estas rutinas establecen cada proceso dentro de una mall. Cada malla `BLACS` está definida en un contexto (que representa su propio universo de paso de mensajes) , de este modo no se interfieren distintos contextos o distintas configuraciones de mallas. Esta rutina puede ser utilizada en repetidas ocasiones para definir contextos/mallas adicionales.

Esta rutina de creación de una malla establece variables internas a las BLACS que no deberán ser utilizadas antes de la llamada a la función `gridinit`.

Hemos de indicar que estas rutinas mapean a los procesos en una malla: los procesos no se crean de forma dinámica. En la mayoría de sistemas paralelos, los procesos no se crean dinamicamente y éstos se crean cuando el usuario arranca el ejecutable.

Esta rutina permite al usuario establecer los procesos en una malla de una manera arbitraria. `usermap(i,j)` establece a un determinado proceso a situarse en i,j en la malla. En los sistemas distribuido, este número de proceso estará simplemente definido por una maquina con un identificador entre 0 y `nprocs-1`. `gridmap` no es recomendable para el usuario inexperto, puesto que `gridinit` es mucho mas simple. `gridinit` simplifica las las funciones de `gridmap` donde los primeros `nprow * npcol` procesos son mapeados dentro de la malla actual en un order natural por filas. `gridmap` permite al usuario avanzado aprovechar algunas caraterísticas del sistema actual.

`gridmap` abre el camino para el multimallado (*multigridding*): el usuario puede separar sus nodos en mallas arbitrarias, unirlas en una misma y luego separarlas en dos nuevas mallas. Mediante `gridmap` tambien podremos establecer mallas arbitrarias o submallas.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - '`ictxt`': Indica el identificador de contexto para ser utilizado en la malla.
  - '`usermap`': Matriz de dimensiones (ldumap,npcol) indicando la identificacion proceso a malla.
  - '`ldumap`': dimension de la matriz usermap
  - '`nprow`': Indica cuantas filas se han de configurar en la malla de procesos.
  - '`npcol`': Indica cuantas columnas se han de configurar en la malla de procesos.
- `Parametros de Salida`
  - '`ictxt`': Entero de salida que indica el contexto BLACS que se ha creado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ldumap=2
usermap=range(0,ldumap*npcol)
ictxt=0
ictxt=PyBLACS.gridmap(ictxt,usermap,ldumap,nprow,npcol)
PyBLACS.gridexit(ictxt)
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgridmap.py
[0, 1, 2, 3]
[0, 1, 2, 3]
[0, 1, 2, 3]
[0, 1, 2, 3]
```

7.3. Destrucción

En este capítulo nos centraremos en las principales rutinas de liberación o destrucción. Este tipo de tareas se realizan de forma automatica en algunas de las rutinas vistas en el capítulo 9 de PyScaLAPACK. Sin embargo, creemos conveniente que el usuario tenga acceso a estas rutinas por si deseara realizar sus propios calculos y necesita de algun tipo de herramienta para intercambiar datos relacionados con el álgebra lineal entre diferentes procesos en una computación en paralelo.

7.3.1. abort

`abort([ictxt,errornum])`

Cuando occurre algún error importante, el usuario puede necesitar abortar todos los procesos. ésta es la razón de la existencia de `abort`. Hemos de destacar que los dos parámetros son de entrada, pero sólo se utilizan para imprimir el mensaje de error. El contexto `ictxt` puede ser cualquiera. Esta rutina mata todos los procesos `BLACS`, pero no solo aquellos confinados a un particular contexto.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - '`ictxt`': Indica el identificador de contexto utilizado en la malla.
  - '`errornum`': Número del error definido por el usuario.
- `Parametros de Salida (no tiene)`

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
PyBLACS.abort(ictxt)
```

7.3.2. gridexit

`gridexit(ictxt)` Los contextos consumen recursos, y es posible que el usuario desee liberarlos cuando ya no le sean necesarios. Despues de la liberación de los recursos, el contexto ya no existe y puede ser reutilizado si se define un nuevo contexto.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - '`ictxt`': Indica el identificador de contexto para ser utilizado en la malla.

- Parametros de Salida (no tiene)

Ejemplos en la utilización de esta rutina los podemos ver en prácticamente la totalidad de ejemplos de las rutinas de este capítulo, puesto que cada vez que inicialicemos una malla, es decir, un contexto tendremos que liberar antes de finalizar el script.

7.3.3. exit

`exit([ictxt])`

Esta rutina debe ser llamada cuando un proceso ha finalizado todo su uso de las BLACS. ésta libera todos los contextos y libera la memoria que las PyBLACS han utilizado.

Las características de cada uno de los parámetros de entrada y salida son:

- Parametros de Entrada
  - 'ictxt': (opcional) Indica el identificador de contexto de la malla que va a ser liberada. En el caso de no proporcionar ningún valor tomaremos el valor por contexto del sistema.
- Parametros de Salida (no tiene)

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

7.3.4. freebuff

`freebuff([ictxt,wait])`

Las PyBLACS tienen al menos un buffer interno utilizado para empaquetar los mensajes. (el número de buffers internos dependen de la plataforma en la que son ejecutadas). En sistemas donde la memoria es escasa, mantener este buffer puede ser caro. Llamar a esta rutina libera?a el buffer de las PyBLACS. De todos modos, la próxima llamada la una rutina de comunicación que requiera empaquetado provocará que el buffer sea realocado.

El parámetro wait determina qué han de esperar las PyBLACS para completar o no las operaciones no-bloqueantes. Si wait=0, las PyBLACS liberarán los buffers que no estén en espera. Si wait<>0, las PyBLACS liberarán todos los buffers internos, incluso aunque ls operaciones no bloqueantes hayan sido completadas.

Las características de cada uno de los parámetros de entrada y salida son:

- Parametros de Entrada
  - wait: Indica si espera (o no) a las operaciones no bloqueantes.
- Parametros de Salida (no tiene)

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
PyBLACS.freebuff(ictxt,1)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

7.4. Informativas

En este capítulo nos centraremos en las principales rutinas informativas, es decir, aquellas que nos proporcionan determinada información de la configuración actual de las PyBLACS. Este conjunto de rutinas son útiles para la identificación de cada uno de los procesos dentro de la malla en el contexto que se le solicite. A continuación describiremos la interfaz de cada una de estas rutinas y mostraremos un ejemplo en su uso.

7.4.1. gridinfo

`nprow,npcol,myrow,mycol=gridinfo([ictxt])`

La rutina `gridinfo` devuelve información relativa a la malla actual. Si el identificador de contexto no es un valor válido todos los valores se devuelven como '`-1`'.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `ictxt=0`:(opcional) El identificador de contexto que nos indicará la configuración de la malla que estamos consultando. Este parámetro es opcional, en el caso de no especificarse obtendremos la malla por defecto inicalizada en el sistema.
- `Parametros de Salida`
  - `nprow`: Número de filas en la malla de procesos
  - `npcol`: Número de columnas en la malla de procesos
  - `myrow`: Número de la fila en la que se encuetra el proceso dentro de la malla.
  - `mycol`: Número de la columna en la que se encuetra el proceso dentro de la malla.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if ictxt<>-1:
        print "Soy ",iam,". I'm in the grid [",myrow,",",mycol,"]"
else:
        print "Soy ",iam,".I'm not in the grid. ictxt=",ictxt
PyBLACS.gridexit(ictxt)
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 6 mpipython exPyblacsgridinfo.py
Soy  5 .I'm not in the grid. ictxt= -1
Soy  4 .I'm not in the grid. ictxt= -1
Soy  1 . I'm in the grid [ 0 , 1 ]
Soy  3 . I'm in the grid [ 1 , 1 ]
Soy  2 . I'm in the grid [ 1 , 0 ]
Soy  0 . I'm in the grid [ 0 , 0 ]
```

7.4.2. pcoord

```
prow,pcol = pcoord(pnum[,ictxt])
```

Dado un intentificador de proceso, la rutina `pcoord` devuelve las coordenadas en las que se encuentra dicho proceso dentro de la malla.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `ictxt=0`:(opcional) El identificador de contexto que nos indicará la configuración de la malla que estamos consultando. Este parámetro es opcional, en el caso de no especificarse obtendremos la malla por defecto inicalizada en el sistema.
- `Parametros de Salida`
  - `prow`: Número de la fila en la que se encuentra el proceso `pnum` dentro de la malla.
  - `pcol`: Número de la columna en la que se encuentra el proceso `pnum` dentro de la malla.

En la rutina siguiente (`pnum`, 7.4.3), podemos observar un ejemplo de la utilización de ambas rutinas.

7.4.3. pnum

```
num = pnum(prow,pcol,[ictxt])
```

Dadas unas coordenadas dentro de una malla de procesos, la rutina `pnum` devuelve el identificador del proceso que se encuentra en esa posición.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `prow`: Número de la fila en la que se encuentra el proceso.
  - `pcol`: Número de la columna en la que se encuentra el proceso.
  - `ictxt=0`:(opcional) El identificador de contexto que nos indicará la configuración de la malla que estamos consultando. Este parámetro es opcional, en el caso de no especificarse obtendremos la malla por defecto inicalizada en el sistema.
- `Parametros de Salida`
  - `num`: Identificador del proceso.

A continuación podemos ver un ejemplo donde se muestra la utilización de distintas funciones como `pnum`, `pcoord`, `gesd2d` y `gerv2d`.

```
import PyACTS.PyBLACS as PyBLACS
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if iam==0:
        for i in range(0,nprow):
                for j in range(0,npcol):
                        if i<>0 and j<>0:
                                icaller=[0]
                                icaller=PyBLACS.gerv2d(icaller,i,j)
                                print icaller

                                if prow==i and pcol==j:
                                        print "Id:",icaller,"-->[",prow, ",",pcol,"]"
else:
        mynum=[icaller]
        PyBLACS.gesd2d(mynum,0,0)
PyBLACS.gridexit()
```

El resultado de la ejecución de este script es:

7.5. Envío

En este capítulo nos centraremos en las principales rutinas de envío de datos. Este tipo de tareas se realizan de forma automatica en algunas de las rutinas vistas en el capítulo 9 de PyScaLAPACK. Sin embargo, creemos conveniente que el usuario tenga acceso a estas rutinas por si deseara realizar sus propios calculos y necesita de algun tipo de herramienta para intercambiar datos relacionados con el álgebra lineal entre diferentes procesos en una computación en paralelo.

7.5.1. gesd2d

`gesd2d(a,rdest,cdest,[ictxt,llda])`

Esta rutina envía la matriz que se le pasa por parámetro al proceso destino descrito mediante las coordenadas `(cdest,rdesc)` dentro de la malla de procesos. Esta rutina es localmente bloqueante, es decir , retornará de su llamada incluso si la correspondiente recepción no ha sido realizada.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `cdest`: Fila en la que se encuentra el proceso destinatario del mensaje.
  - `rdest`: Columna en la que se encuentra el proceso destinatario del mensaje.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
- `Parametros de Salida (no tiene)`

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
size=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if myrow==0 and mycol==0:
        a=reshape(range(size),[size,1])
        print "[",myrow,",",mycol,"] sends a=",transpose(a)
        PyBLACS.gesd2d(a,0,1)
        PyBLACS.gesd2d(a,1,0)
        PyBLACS.gesd2d(a,1,1)
else:
        a=zeros([size,1])
        a=PyBLACS.gerv2d(a,0,0)
        print "[",myrow,",",mycol,"] receives a=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgesdrv2d.py
[ 0 , 0 ] sends a= [ [0 1 2 3]]
[ 0 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 0 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
```

Tal y como se puede observar en el ejemplo, en el nodo principal ([0,0])se generan los datos (o en otros casos se pueden leer de un fichero), y éste los envía al resto de procesos. Las rutinas de envío y recepción son considerablemente inferiores a las análogas en BLACS puesto que no hemos de indicar parámetros relativos al tamaño las matrices, ya que éstas características se encuentran incluidas como propiedades del `Numarray` que se le pasa como parámetro. Un detalle significativo que se debe señalar reside en la necesidad de definir la variable donde se va a recibir el dato para poder hacer reserva de los recursos de memoria o bien reutilizar una variable anterior. Esta necesidad de definir, el espacio de memoria se puede realizar simplemente definiendo una matriz de ceros del tamaño indicado. El comando utilizado para esta finalidad sería `a=zeros([size,1])` y computacionalmente no es costoso. Se ha preferido esta opción a la opción que la rutina receptora genere una matriz del tamaño indicado para así poder reutilizar el espacio de memoria de otras variables que ya no son útiles.

7.5.2. trsd2d

`trsd2d(a,rdest,cdest,[ictxt,llda])`

Esta rutina envia la matriz que se le pasa por parametro al proceso destino descrito mediante las coordenadas `(cdest,rdesc)` dentro de la malla de procesos.La matriz en este caso es de tipo triangular, de esto modo usaremos los parametros `uplo`, y `diag` para describir las caracteristicas de la matriz triangular.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.

- `cdest`: Fila en la que se encuentra el proceso destinatario del mensaje.
- `rdest`: Columna en la que se encuentra el proceso destinatario del mensaje.
- `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
- `uplo`: Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
  - `uplo='U'`: (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
  - `uplo='L'`: Se obtienen los resultados en la diagonal principal y por debajo de ella.
- `diag`: Indicará si la matriz es unitaria triangular o no.
  - `diag='N'`: (Valor por defecto). La matrix no es unitaria triangular
  - `diag='U'`: La matrix es unitaria triangular
- `llda`: Leading dimension de a.

- `Parametros de Salida (no tiene)`

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
size=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if myrow==0 and mycol==0:
        a=reshape(range(size),[size,1])
        print "[",myrow,",",mycol,"] sends a=",transpose(a)
        PyBLACS.gesd2d(a,0,1)
        PyBLACS.gesd2d(a,1,0)
        PyBLACS.gesd2d(a,1,1)
else:
        a=zeros([size,1])
        a=PyBLACS.gerv2d(a,0,0)
        print "[",myrow,",",mycol,"] receives a=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgesdrv2d.py
[ 0 , 0 ] sends a= [ [0 1 2 3]]
[ 0 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 0 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
```

7.6. Recepción

En este capítulo nos centraremos en las principales rutinas de recepción de datos. Este tipo de tareas se realizan de forma automatica en algunas de las rutinas vistas en el capítulo 9 de PyScaLAPACK. Sin embargo, creemos conveniente que el usuario tenga acceso a estas rutinas por si deseara realizar sus propios calculos y necesita de algun tipo de

herramienta para intercambiar datos relacionados con el álgebra lineal entre diferentes procesos en una computación en paralelo.

Por otro lado, algunas de estas rutinas ya se han visto en el apartado anterior referente a las rutinas de envio de datos (7.5)

7.6.1. gerv2d

`a=gerv2d(a,rsrc,csrc,[ictxt,llda])`

Esta rutina recibe la matriz que se le pasa por parametro desde el proceso descrito mediante las coordenadas `(csrc,rsrc)` dentro de la malla de procesos. Esta rutina es localmente bloqueante, por ejemplo, retornará incluso si la correspondiente recepción no ha sido realizada.

Un detalle importante a tener en cuenta es que a pesar de ser una rutina de recepción, deberemos pasar como parámetro a esta rutina la matriz `a`. Esta matriz será una matriz de ceros (por ejemplo), con las dimensiones adecuadas de la matriz que esperemos recibir.

El resultado será una matriz de las dimensiones de la matriz de entrada con sus elementos correspondientes a los elementos recibidos.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz de tipo de dato `Numeric` de ceros donde copiaremos la matriz recibida.
  - `csrc`: Fila en la que se encuentra el proceso fuente del mensaje.
  - `rsrc`: Columna en la que se encuentra el proceso fuente del mensaje.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
- `Parametros de Salida`
  - `a`: Matriz de datos recibida.

En el apartado 7.7.1 ya hemos mostrado un ejemplo en la utilización de esta rutina por lo que no creemos conveniente repetir otro ejemplo.

7.6.2. trrv2d

`a=trrv2d(a,rsrc,csrc,[ictxt,llda,diag,llda])`

Esta rutina recibe la matriz que se le pasa por parametro desde proceso fuente descrito mediante las coordenadas `(csrc,rsrc)` dentro de la malla de procesos.La matriz en este caso es de tipo triangular, de esto modo usaremos los parametros `uplo`, y `diag` para describir las caracteristicas de la matriz triangular.

Un detalle importante a tener en cuenta es que a pesar de ser una rutina de recepción, deberemos pasar como parámetro a esta rutina la matriz `a`. Esta matriz será una matriz de ceros (por ejemplo), con las dimensiones adecuadas de la matriz que esperemos recibir.

El resultado será una matriz de las dimensiones de la matriz de entrada con sus elementos correspondientes a los elementos recibidos. Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz de tipo de dato `Numeric` de ceros donde copiaremos la matriz recibida.

- `cdest`: Fila en la que se encuentra el proceso fuente del mensaje.
- `rdest`: Columna en la que se encuentra el proceso fuente del mensaje.
- `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
- `uplo`: Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
  - `uplo='U'`: (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
  - `uplo='L'`: Se obtienen los resultados en la diagonal principal y por debajo de ella.
- `diag`: Indicará si la matriz es unitaria triangular o no.
  - `diag='N'`: (Valor por defecto). La matrix no es unitaria triangular
  - `diag='U'`: La matrix es unitaria triangular
- `llda`: Leading dimension de a.

- `Parametros de Salida`
  - `a`: Matriz de datos recibida.

En el apartado 7.5.2 ya hemos mostrado un ejemplo en la utilización de esta rutina por lo que no creemos conveniente repetir otro ejemplo.

7.7. Difusión

En este capítulo nos centraremos en las principales rutinas de difusion (*Brodcast*) de datos.

7.7.1. gebs2d

`gesd2d(a[,ictxt,scope,top,llda])`

Esta rutina inicia la difusión a traves del grupo de destinatarios. Todos los demás procesos que pertenecen al grupo de destino deberán llamar a la rutina de recepción de la difusión (`gebr2d`). Finalmente, todos los procesos pertenecientes al grupo destinatario recibirán la matriz `a`. La difusión es localmente bloqueante, es decir , que no se garantiza que el proceso vuelva de una llamada a broadcast hasta que todos los procesos hayan llamado a las rutinas apropiadas (`gebs2d` o `gebr2d`).

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión:
    - `top=' '`: (Valor por defecto). Topología dependiente del sistema por defecto

- top='I': Anillo incremental
- top='D': Anillo decremental
- top='H': Hipercubo (mínimo *spanning tree*)
- top='S': Anillo *split*
- top='F': Conectividad total
- top='M': Nodos divididos en 'I' anillos incrementales, donde 'I' es un conjunto obtenido de la llamada a `PyBLACS.set`
- top='T': Arbol de difusión con `nbranches=I`, donde 'I' es un conjunto obtenido de la llamada a `PyBLACS.set`
- top='1': Arbol de difusión con `nbranches=1`
- top='2': Arbol de difusión con `nbranches=2`
- ...
- top='9': Arbol de difusión con `nbranches=9`

- `llda`: Leading dimension de a.

- `Parametros de Salida (no tiene)`

En el ejemplo que mostramos a continuación, realizaremos la misma funcionalidad que en el ejemplo del apartado `gesd2d` 7.7.1 pero utilizando las rutinas de difusión y recolección.

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
size=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if myrow==0 and mycol==0:
        a=reshape(range(size),[size,1])
        print "[",myrow,",",mycol,"] brodcast a=",transpose(a)
        PyBLACS.gebs2d(a,top='F')
else:
        a=zeros([size,1])
        a=PyBLACS.gebr2d(a,0,0,top='F')
        print "[",myrow,",",mycol,"] receives a=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgebsr2d.py
[ 0 , 0 ] brodcast a= [ [0 1 2 3]]
[ 0 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 0 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
```

7.7.2. trbs2d

`trbs2d(a[,ictxt,scope,top,uplo,diag,llda])`

Esta rutina inicia la difusión a traves del grupo de destinatarios. Todos los demás procesos que pertenecen al grupo de destino deberán llamar a la rutina de recepción de la difusión (`gebr2d`). Finalmente, todos los procesos pertenecientes al grupo destinatario recibirán la matriz `a`. La difusión es localmente bloqueante, es decir , que no se garantiza que el proceso vuelva de una llamada a broadcast hasta que todos los procesos hayan llamado a las rutinas apropiadas (`trbs2d` o `trbr2d`). En este caso las matrices a enviar son de tipo trapezoidal.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión (ver rutina `gebs2d`,**??**)
  - `uplo`: Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
    - `uplo='U'`: (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
    - `uplo='L'`: Se obtienen los resultados en la diagonal principal y por debajo de ella.
  - `diag`: Indicará si la matriz es unitaria triangular o no.
    - `diag='N'`: (Valor por defecto). La matrix no es unitaria triangular
    - `diag='U'`: La matrix es unitaria triangular
  - `llda`: Leading dimension de a.
- `Parametros de Salida (no tiene)`

En el ejemplo que mostramos a continuación, realizaremos la misma funcionalidad que en el ejemplo del apartado `trsd2d` 7.5.2 pero utilizando las rutinas de difusión y recolección.

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
size=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
if myrow==0 and mycol==0:
        a=reshape(range(size),[size,1])
        print "[",myrow,",",mycol,"] brodcast a=",transpose(a)
        PyBLACS.trbs2d(a,top='F')
else:
        a=zeros([size,1])
        a=PyBLACS.trbr2d(a,0,0,top='F')
        print "[",myrow,",",mycol,"] receives a=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgebsr2d.py
[ 0 , 0 ] brodcast a= [ [0 1 2 3]]
[ 0 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 0 ] receives a= [ [ 0.  1.  2.  3.]]
[ 1 , 1 ] receives a= [ [ 0.  1.  2.  3.]]
```

7.8. Recolección

En el presente capítulo describiremos las principales rutinas de recolección de datos. Estas rutinas ya han sido introducidas y utilizadas en el apartado de difusion (*Brodcast*) por lo que no creemos necesario utilizar ejemplos en este apartado para ilustrar su utilización.

7.8.1. gebr2d

`a=gebr2d(a,irsrc,icsrc[,ictxt,scope,top,lda])`

Esta rutina recibe y participa en una difusión a traves del grupo de destinatarios. Todos los demás procesos que pertenecen al grupo de destino deberán llamar a la rutina de recepción de la difusión (`gebr2d`). Finalmente, todos los procesos pertenecientes al grupo destinatario recibirán la matriz `a`.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `irsrc`: El número de la fila del procesos que llama a la Difusión *Broadcast / send*.
  - `crsrc`: El número de columna del procesos que llama a la Difusión *Broadcast / send*.
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión.(ver apartado `gebs2d`)
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
- `Parametros de Salida`
  - `a`: Matriz de datos recibida.

En la descripción del apartado `gebs2d` (**??**) podemos ver ejemplos en la utilización de esta rutina.

7.8.2. trbr2d

`a=trbr2d(a,irsrc,icsrc[,ictxt,uplo,diag,scope,top,lda])`

Esta rutina recibe y participa en una difusión a traves del grupo de destinatarios. Todos los demás procesos que pertenecen al grupo de destino deberán llamar a la rutina de recepción de la difusión (`trbr2d`). Finalmente, todos los procesos pertenecientes al grupo destinatario recibirán la matriz `a`.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `irsrc`: El número de la fila del procesos que llama a la Difusión *Broadcast / send*.
  - `crsrc`: El número de columna del procesos que llama a la Difusión *Broadcast / send*.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `uplo`: Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
    - `uplo='U'`: (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
    - `uplo='L'`: Se obtienen los resultados en la diagonal principal y por debajo de ella.
  - `diag`: Indicará si la matriz es unitaria triangular o no.
    - `diag='N'`: (Valor por defecto). La matrix no es unitaria triangular
    - `diag='U'`: La matrix es unitaria triangular
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión.(ver apartado `gebs2d`)
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
- `Parametros de Salida`
  - `a`: Matriz de datos recibida.

En la descripción del apartado `trbs2d` (7.7.2) podemos ver ejemplos en la utilización de esta rutina.

7.9. Operaciones Combinadas

Por último, dentro de las librerías `PyBLACS` tenemos disponibles un conjunto de rutinas que además de implementar funciones de comunicaciones realizan operaciones que suelen ser útiles cuando se está trabajando con matrices. Estas operaciones son tales como el cálculo del mínimo (`gamn2d`), del máximo (`gamx2d`) o bien la suma de diferentes matrices o vectores en los diferentes procesos (`gsum2d`). Describiremos cada una de estas rutinas indicando sus parametros principales y opcionales. Por otro lado, mostraremos ejemplos en la utilización de cada uno de ellos.

7.9.1. gsum2d

`a=gsum2d(a,rdest,cdest[,ictxt,scope,top,lda])`

Mediante la llamada a esta rutina, cada elemento del vector /matriz `a` es sumado con los correspondientes elementos de los demás matrices de los demás procesos. La combinación puede ser globalmente bloqueante , de este modo no se devuelve una respuesta hasta que todos los procesos han llamado a esta rutina. Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `cdest`: Fila en la que se encuentra el proceso destinatario del mensaje.
  - `rdest`: Columna en la que se encuentra el proceso destinatario del mensaje.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión:
- `Parametros de Salida`
  - `a`: Contiene el resultado en aquel proceso que se haya elegido como destino.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
from RandomArray import *
from random import Random
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
irsrc,icsrc=0,0
g=Random(iam)
n=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
x=iam*reshape(range(n),[n,1])
print "x[",myrow,",",mycol,"]=",transpose(x)
a=PyBLACS.gsum2d(x,irsrc,icsrc)
if myrow==irsrc and mycol==icsrc:
        print "sum=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgsum2d.py
x[ 0 , 1 ]= [ [0 1 2 3]]
x[ 0 , 0 ]= [ [0 0 0 0]]
x[ 1 , 0 ]= [ [0 2 4 6]]
x[ 1 , 1 ]= [ [0 3 6 9]]
sum= [ [  0.   6.  12.  18.]]
```

7.9.2. gamx2d

`a,ra,ca=gamx2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`

Esta rutina obtiene los valores máximos comparando cada elemento de la matriz con los demás elementos de las matrices de cada proceso en la misma posición. El resultado final se almacena en la matriz del proceso con coordenadas [rdest,cdest]. El resultado es devuelto en valor absoluto. En el caso de utilizar numeros complejos se calculará la norma. La ejecución de esta rutina puede bloquear globalmente, por tanto todos los procesos que intervienen deberán ejecutar esta rutina.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `cdest`: Fila en la que se encuentra el proceso destinatario del mensaje.
  - `rdest`: Columna en la que se encuentra el proceso destinatario del mensaje.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
  - `scope`: Indica el grupo de destino de la difusíon.
    - `scope='A'`: (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - `scope='R'`: Se envían los datos a los procesos de la misma fila.
    - `scope='C'`: Se envían los datos a los procesos de la misma columna.
  - `top`: Topología de la difusión:
- `Parametros de Salida`
  - `a`: Contiene el resultado en aquel proceso que se haya elegido como destino.
  - `ra`: Si `rflag=-1` esta matriz no será referenciado y por tanto no existirá. Sin embargo, si utilizaramos una matriz de enteros (de tamaño `rcflags` x `n`) indicando el indice de fila del proceso que proporcionó el máximo.
  - `ca`: Si `rflag=-1` esta matriz no será referenciado y por tanto no existirá. Sin embargo, si utilizaramos una matriz de enteros (de tamaño `rcflags` x `n`) indicando el indice de columna del proceso que proporcionó el máximo.
  - `ca`: Matriz de datos recibida.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
from RandomArray import *
from random import Random
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
irsrc,icsrc=0,0
g=Random(iam)
n=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
x=zeros([n,1],Float)
for i in range(0,n):
        x[i]=100*g.random()
print "x[",myrow,",",mycol,"]=",transpose(x)
a,ra,ca=PyBLACS.gamx2d(x,irsrc,icsrc)
if myrow==irsrc and mycol==icsrc:
        print "max=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgamx2d.py
x[ 1 , 1 ]= [ [  3.38789388  79.33674928  68.89134666  67.87649609]]
x[ 0 , 1 ]= [ [  2.25802504  86.12917718  30.39651697  85.26061846]]
x[ 1 , 0 ]= [ [  2.82295946  82.73296323  49.64393182  76.56855727]]
x[ 0 , 0 ]= [ [  1.69309062  89.52539112  11.14910212  93.95267964]]
max= [ [  3.38789392  89.52539062  68.89134979  93.9526825 ]]
```

7.9.3. gamn2d

`a,ra,ca=gamn2d(a,rdest,cdest[,ictxt,scope,top,lda,rcflag])`

La columna de procesos coordina el proceso que debería recibir los resultados. Si `rdest=-1`, entonces `rdest` es ignorado. Esta rutina obtiene los valores mínimos comparandolos para cada elemento con los demas elementos de las matrices de cada proceso en la misma posicion. El resultado es devuelto en valor absoluto. En el caso de utilizar numeros complejos se calculará la norma. La ejecución de esta rutina puede bloquear globalmente, por tanto todos los procesos que intervienen deberán ejecutar esta rutina.

Las características de cada uno de los parámetros de entrada y salida son:

- `Parametros de Entrada`
  - `a`: Matriz en tipo de dato `Numeric` a ser enviada.
  - `cdest`: Fila en la que se encuentra el proceso destinatario del mensaje.
  - `rdest`: Columna en la que se encuentra el proceso destinatario del mensaje.
  - `ictxt`: Indicador del contexto, parametro opcional. En el caso de no indicarlo, se utilizará el identificador de contexto por defecto.
  - `llda`: Leading dimension de a.
  - `scope`: Indica el grupo de destino de la difusíon.

    - scope='A': (Valor por defecto). Correspondiente a *All*. Todos los procesos de la malla recibirán la matriz
    - scope='R': Se envían los datos a los procesos de la misma fila.
    - scope='C': Se envían los datos a los procesos de la misma columna.
  - top: Topología de la difusión:

- Parametros de Salida
  - a: Contiene el resultado en aquel proceso que se haya elegido como destino.
  - ra: Si rflag=-1 este array no será referenciado y por tanto no existirá. Sin embargo, si utilizaramos una matriz de enteros (de tamaño rcflags x n) indicando el indice de fila del proceso que proporcionó el máximo.
  - ca: Si rflag=-1 este array no será referenciado y por tanto no existirá. Sin embargo, si utilizaramos una matriz de enteros (de tamaño rcflags x n) indicando el indice de columna del proceso que proporcionó el máximo.
  - ca: Matriz de datos recibida.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
import PyACTS.PyBLACS as PyBLACS
from Numeric import *
from RandomArray import *
from random import Random
iam,nprocs=PyBLACS.pinfo()
nprow,npcol=2,2
irsrc,icsrc=0,0
g=Random(iam)
n=4
ictxt=PyBLACS.gridinit(nprow,npcol)
nprow,npcol,myrow,mycol=PyBLACS.gridinfo()
x=zeros([n,1],Float)
for i in range(0,n):
        x[i]=100*g.random()
print "x[",myrow,",",mycol,"]=",transpose(x)
a,ra,ca=PyBLACS.gamn2d(x,irsrc,icsrc)
if myrow==irsrc and mycol==icsrc:
        print "min=",transpose(a)
PyBLACS.gridexit(ictxt)
PyBLACS.exit()
```

El resultado de la ejecución de este script es:

```
vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyblacsgamn2d.py
x[ 0 , 1 ]= [ [  2.25802504  86.12917718  30.39651697  85.26061846]]
x[ 1 , 1 ]= [ [  3.38789388  79.33674928  68.89134666  67.87649609]]
x[ 1 , 0 ]= [ [  2.82295946  82.73296323  49.64393182  76.56855727]]
x[ 0 , 0 ]= [ [  1.69309062  89.52539112  11.14910212  93.95267964]]
min= [ [  1.69309068  79.33674622  11.14910221  67.87649536]]
```

CAPÍTULO

EIGHT

PyPBLAS

Dentro de la distribución estándar de ScaLAPACK, se encuentran el conjunto de rutinas denominado PBLAS (*Parallel Basic Linear Algebra Subprograms*). Este conjunto de rutinas resuelven las principales operaciones algebraicas relacionadas con escalares, vectores y matrices. Se agrupan en tres niveles dependiendo de la complejidad de las operaciones implementadas:

- `Nivel 1`: Operaciones en las que intervienen únicamente vectores y escalares.
- `Nivel 2`: Operaciones entre vectores y matrices.
- `Nivel 3`: Operaciones entre matrices.

Esta agrupación se ha llevado a cabo en la distribución estandar de las PBLAS. En las rutinas que mostramos a continuación hemos intentado respetar al máximo la nomenclatura de cada una de las rutinas así como la nomenclatura en los parametros utilizados. De este modo, cualquier persona con conocimientos básicos en el funcionamiento de PBLAS, le resultaría muy facil poder hacer uso de las rutinas implementadas en PyPBLAS puesto que muchas de las operaciones se automatizan resultando un proceso mucho mas sencillo como veremos en los diferentes ejemplos de este capítulo.

En el presente capítulo agruparemos las rutinas de PyPBLAS por niveles (1,2,3), sin embargo en el nombre de la rutina no se especifica ninguna relación con el nivel al que pertenece (igual que sucede con la distribución PBLAS). En nuestro caso, todas las rutinas que se corresponden con la colección PBLAS tienen el comienzo idetificativo de dicha colección, es decir, las funciones se llamarán `PyPBLAS.*` desde cualquier script de Python. Se ha de tener en cuenta que las rutinas `PyPBLAS.*` se encuentran en el módulo `PyScaLAPACK`, por lo que deberemos importar este módulo mediante `import PyScaLAPACK` o `from PyScaLAPACK import *`. El término que hemos señalado como `*` se corresponde con el nombre de la rutina que recibe dentro de la distribución PBLAS, de este modo conseguimos homogeneidad en la nomenclatura de nuestra distribución y facilitamos la migaración en la utilización de PBLAS a PyPBLAS o viceversa.

Antes de comenzar a ver el conjunto de rutinas importadas, se ha de tener en cuenta el nombre de las rutinas en PBLAS es dependiente del tipo de dato con el que vayamos a tratar, de este modo en vez de la letra v que aparecen en los nombres de las rutinas de PBLAS se deberá sustituir por la letra correspondiente dependiendo del tipo de dato utilizado:

Letra	Tipo
`S`	Dato real de precision simple
`D`	Dato real de precision doble
`C`	Dato complejo de precision simple
`Z`	Dato complejo de precision doble

Sin embargo, no vamos a necesitar realizar esta distinción en las rutinas de PyPBLAS puesto que llamaremos a la misma rutina de Python y de forma interna se determina el tipo de de dato que ésta contiene y hará uso de la rutina correspondiente de modo interno y ajeno al usuario. De este modo conseguimos facilitar el trabajo en gran medida a un investigador ajeno al uso de las librerías PBLAS.

8.1. Referencia rápida de las librerías PyPBLAS

Operación	**Rutina**
	*N*IVEL 1
$x \longleftrightarrow y$	`x,y= pvswap(x,y)`
$\alpha x \rightarrow x$	`x= pvscal(alpha,x)`
$x \rightarrow y$	`y= pvcopy(x)`
$\alpha x + y \rightarrow y$	`y= pvaxpy(alpha,x,y)`
$x^T y \rightarrow y$	`dot= pvdot(x,y)`
$x^T y \rightarrow y$	`dot= pvdotu(x,y)`
$x^H y \rightarrow y$	`dot= pvdotc(x,y)`
$\|x\|2 \rightarrow y$	`nrm2= pvnrm2(x)`
$\|re(x)\|2 + \|img(x)\|2 \rightarrow asum$	`asum= pvasum(x)`
$1^{st}k \ni \|re(xk)\| + \|img(xk)\| \rightarrow indx$	`amax,indx= pvamax(x)`
$max\|re(xi)\| + \|img(xi)\| \rightarrow amax$	
	*N*IVEL 2
$\alpha \cdot op(A) \cdot x + \beta \cdot y \rightarrow y$	`y= pvgemv(alpha,a,x,beta,y[,trans])`
$\alpha \cdot A \cdot x + \beta \cdot y \rightarrow y$	`y= pvhemv(alpha,a,x,beta,y[,uplo])`
$\alpha \cdot A \cdot x + \beta \cdot y \rightarrow y$	`y= pvsymv(alpha,a,x,beta,y)`
$A \cdot x \rightarrow x; A^T \cdot x \rightarrow x; A^H \cdot x \rightarrow x$	`x= pvtrmv(a,x,[uplo,trans,diag]`
$A^{-1} \cdot x \rightarrow x; A^{-T} \cdot x \rightarrow x; A^{-H} \cdot x \rightarrow x$	`x= pvtrsv(a,x,[uplo,trans,diag])`
$\alpha \cdot x \times y^T + A \rightarrow A$	`a= pvger(alpha,x,y,a)`
$\alpha \cdot x \times y^T + A \rightarrow A$	`a= pvgeru(alpha,x,y,a)`
$\alpha \cdot x \times y^H + A \rightarrow A$	`a= pvgerc(alpha,x,y,a)`
$\alpha \cdot x \times x^H + A \rightarrow A$	`a= pvher(alpha,x,a,[uplo])`
$\alpha \cdot x \times y^H + y(alpha \cdot x)^H + A \rightarrow A$	`a= pvher2(alpha,x,y,a,[uplo])`
$\alpha \cdot x \times x^T + A \rightarrow A$	`a= pvsyr(alpha,x,a,[uplo='U'])`
$\alpha \cdot x \times y^T + y(\alpha \cdot x)^T + A \rightarrow A$	`a= pvsyr2(alpha,x,y,a,[uplo])`
	*N*IVEL 3
$\alpha \cdot op(A) \times op(B) + \beta \cdot C \rightarrow C$	`c= pvgemm(alpha,a,b,beta,c,[transa,transb])`
$\alpha \cdot A \times B + \beta \cdot C \rightarrow C; \alpha \cdot B \times A + \beta \cdot C \rightarrow$	`c=pvsymm(alpha,a,b,beta,c[,side,uplo])`
$\alpha \cdot A \times B + \beta \cdot C \rightarrow C; \alpha \cdot B \times A + \beta \cdot C \rightarrow C$	`c=pvhemm(alpha,a,b,beta,c[,side,uplo])`
$\alpha \cdot A \times A^T + \beta \cdot C \rightarrow C; \alpha \cdot A^T \times A + \beta \cdot C \rightarrow C$	`c=pvsyrk(alpha,a,beta,c[,uplo,trans])`
$\alpha \cdot A \times A^H + \beta \cdot C \rightarrow C; \alpha \cdot A^H \times A + \beta \cdot C \rightarrow C$	`c=pvherk(alpha,a,beta,c[,uplo,trans])`
$\alpha \cdot A \times B^T + \alpha \cdot B \times A^T + beta \cdot C \rightarrow C$	`c=pvsyr2k(alpha,a,b,beta,c[,uplo,trans])`
$\alpha \cdot A \times B^H + \alpha \cdot B \times A^H + beta \cdot C \rightarrow C$	`c=pvher2k(alpha,a,b,beta,c[,uplo,trans])`
$\beta \cdot C + \alpha \cdot A^T + \rightarrow C$	`c=pvtran(alpha,a,beta,c)`
$\beta \cdot C + \alpha \cdot A^T + \rightarrow C$	`c=pvtranu(alpha,a,beta,c)`
$\beta \cdot C + \alpha \cdot A^H + \rightarrow C$	`c=pvtranc(alpha,a,beta,c)`
$\alpha \cdot op(A) \times B \rightarrow B; \alpha \cdot B \times op(A) \rightarrow B$	`b=pvtrmm(alpha,a,b[,side,uplo,transa,diagç])`
$\alpha \cdot op(A^-1) \times B \rightarrow B; \alpha B \cdot B \times op(A^-1) \rightarrow B$	`b=pvtrsm(alpha,a,b[,side,uplo,transa,diag])`

8.2. Referencia rápida del módulo PyScaLAPACK

8.3. PyPBLAS nivel 1

8.3.1. pvswap

`x,y= PyPBLAS.pvswap(x,y)`

La funcion `pvswap` intercambia el contenido de los datos de dos vectores que se le introducen como parametros: Es decir, la operación básica que realizan es el intecambio de valores entre los variables de tipo `PyACTS` `x` e `y`.

- `Parametros de Entrada`
  - 'x': Vector de tipo PyACTS
  - 'y': Vector de tipo PyACTS
- `Parametros de Salida`
  - 'x': Vector de tipo PyACTS
  - 'y': Vector de tipo PyACTS

Se ha de tener en cuenta que x e y son dos variables de tipo `PyACTS` que pueden estar distribuidos entre los procesos de la malla de procesos configurada en la incialización del sistema. Por ejempo:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvSWAP"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=reshape(array(range(n)),[n,1])
        y=ones([n,1])
        print "x=",transpose(x)
        print "y=",transpose(y)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
x,y= PyPBLAS.pvswap(x,y)
if PyACTS.ictxt<>-1:
        print "x[",PyACTS.myrow,",",PyACTS.mycol,"]=",x.data
        print "y[",PyACTS.myrow,",",PyACTS.mycol,"]=",y.data
PyACTS.gridexit()
```

El resultado que obtenemos es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvswap.py
Example of using PyPBLAS: PvSWAP
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
x= [ [0 1 2 3 4 5 6 7]]
y= [ [1 1 1 1 1 1 1 1]]
x[ 0 , 0 ]= [0 1 4 5]
y[ 0 , 0 ]= [1 1 1 1]
x[ 1 , 0 ]= [2 3 6 7]
y[ 1 , 0 ]= [1 1 1 1]
x[ 0 , 1 ]= zeros((0,), 'l')
y[ 0 , 1 ]= zeros((0,), 'l')
x[ 1 , 1 ]= zeros((0,), 'l')
y[ 1 , 1 ]= zeros((0,), 'l')
```

En este ejemplo se han utilizado determinadas funciones de incialización de la malla de procesos que intervienen en los cálculos, y de liberación de dicha malla que ya han sido explicados y detallados en el capítulo 6 como: '`PyACTS.gridinit()`' y '`PyACTS.gridexit()`'. En el resultado mostrado por pantalla comprobamos cómo se han intercambiado los valores de `x.data` e `y.data`. Se ha de tener en cuenta en el presente ejemplo, que se ha hecho una impresión por pantalla (mediante '`print "x[",...`') en todos los procesos que pertenecen a la malla de procesos que se ha configurado (en este caso de dimensiones 3*2). Cada uno de los procesos imprime los valores que posea el vector a modo local, es decir el vector en este caso está distribuido en varios de los procesos que componen la malla. En determinadas ocasiones puede ser interesante recoger la totalidad del vector en uno de los procesos, con tal finalidad se ha implementado la función '`PyACTS2Num`' descrita en el capítulo 6. En este ejemplo podríamos haber recogido las variables en el proceso 0 mediante el comando:

```
x_global=PyACTS2Num(x)
```

8.3.2. pvscal

`x= PyPBLAS.pvscal(alpha,x)`

La función '`PyPBLAS.pvscal`' distribuye (si es necesario) y multiplica un vector PyACTS (`x`) por un número escalar real (`alpha`). El resultado es la la variable `x` cuyos datos (`x.data`) se encuentran multiplicados por el escalar correspondiente.

Los parametros de entrada y salida de esta función son los siguientes:

- `Parametros de Entrada`
  - '`alpha`': Escalar PyACTS
  - '`y`': Vector de tipo PyACTS
- `Parametros de Salida`
  - '`x`': Escalar PyACTS
  - '`y`': Vector de tipo PyACTS

Observamos tambien que el escalar conviene ser tratado previamente a modo que todos los procesadores conozcan su valor. Mediante la llamada a `alpha=Scal2PyACTS(alpha,ACTS_lib)` estamos preparando la variable `alpha` para poder ser utilizada como escalar en las rutinas de las PBLAS.

La funcionalidad de los descriptores `x.desc, ..` se describen en el capítulo 5. Un ejemplo de la utilización de esta función sería el siguiente:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvSCAL"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=reshape(array(range(n)),[n,1])
        print "x=",transpose(x)
        alpha=2
else:
        x,alpha=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
alpha=Scal2PyACTS(alpha,ACTS_lib)
#We call PBLAS routine
x= PyPBLAS.pvscal(alpha,x)
if PyACTS.ictxt<>-1:
        print "x[",PyACTS.myrow,",",PyACTS.mycol,"]=",x.data
#We convert PyACTS.Scalapack Array to Numerical Python
x_num=PyACTS2Num(x)
if PyACTS.iread==1:
        print "Resultado Global:"
        print "x'=",transpose(x_num)
PyACTS.gridexit()
```

El resultado que obtenemos sería el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvscal.py
Example of using PyPBLAS: PvSCAL
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
x= [ [0 1 2 3 4 5 6 7]]
x[ 0 , 0 ]= [  0.   2.   8.  10.]
x[ 1 , 0 ]= [  4.   6.  12.  14.]
x[ 0 , 1 ]= zeros((0,), 'f')
x[ 1 , 1 ]= zeros((0,), 'f')
Resultado Global:
x'= [ [  0.   2.   4.   6.   8.  10.  12.  14.]]
```

En este ejemplo vemos que el vector generado es un vector de n elementos (`[0,1,...,n-1]`) donde `n` en este caso vale 8. Podemos observar como cada proceso contiene una parte del resultado de la operación de multiplicación y al ejecutar `x_num=PyACTS2Num(x)` se obtiene la totalidad del vector en el proceso en el que se cumple `PyScaLAPACK.iread=1` (realizando la lectura-escritura en un único proceso).

8.3.3. pvcopy

```
y= PyPBLAS.pvcopy(x)
```

La función '`PyPBLAS.pvcopy`' distribuye (si es necesario) y copia el contenido de un vector `x` en otro vector `y`. Se

ha tener en cuenta que los parámetros de entrada y salida son de tipo PyACTS. Los mínimos parametros de entrada de la función son:

- `Parametros de Entrada`
  - 'x': Vector a ser copiado.
- `Parametros de Salida`
  - 'y': Vector PyACTS donde se ha realizado la copia de x. Es de carácter global, es decir se encuentra distruido entre los procesos de la malla.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvCOPY"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=reshape(array(range(n),Float),[n,1])
        print "x=",transpose(x)
else:
        x=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
#We call PBLAS routine
y= PyPBLAS.pvcopy(x)
if PyACTS.ictxt<>-1:
        print "x[",PyACTS.myrow,",",PyACTS.mycol,"]=",transpose(x.data)
        print "y[",PyACTS.myrow,",",PyACTS.mycol,"]=",transpose(y.data)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvcopy.py
Example of using PyPBLAS: PvCOPY
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
x= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
x[ 0 , 0 ]= [ 0.  1.  4.  5.]
y[ 0 , 0 ]= [ [ 0.  1.  4.  5.]]
x[ 0 , 1 ]= zeros((0,), 'd')
y[ 0 , 1 ]= zeros((0, 4), 'f')
x[ 1 , 0 ]= [ 2.  3.  6.  7.]
y[ 1 , 0 ]= [ [ 2.  3.  6.  7.]]
x[ 1 , 1 ]= zeros((0,), 'd')
y[ 1 , 1 ]= zeros((0, 4), 'f')
```

8.3.4. pvaxpy

`y= PyPBLAS.pvaxpy(alpha,x,y)`

La función '`PyPBLAS.pvaxpy`' suma el vector `y` con el resultado de multiplicar el vector `x` por un escalar `alpha`. La operación se puede resumir como: '`y=y+alpha*x`'. Los vectores '`x`' e '`y`' han de ser tipo `PyACTS`.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`': Numero escalar de tipo real.
  - '`x`': Vector x de tipo PyACTS de la operación descrita.
  - '`y`': Vector y de tipo PyACTS de la operación descrita.
- `Parametros de Salida`
  - '`y`': Vector x de tipo PyACTS , es decir, sus elementos se han distruido entre los elementos de la malla de procesos (ver capítulo 5).

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvAXPY"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=2
        print "alpha=",alpha
        x=(1+1j)*reshape(array(range(n),Numeric.Float),[n,1])
        print "x'=",transpose(x)
        y=(1+1j)*ones([n,1])
        print "y'=",transpose(y)
else:
        x,y,alpha=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
alpha=Scal2PyACTS(alpha,ACTS_lib)
#We call PBLAS routine
y= PyPBLAS.pvaxpy(alpha,x,y)

y_num=PyACTS2Num(y)
if PyACTS.iread==1:
        print "Resultado Global:"
        print "y'=",transpose(y_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvaxpy.py
Example of using PyPBLAS: PvAXPY
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 2
x'= [ [ 0.+0.j  1.+1.j  2.+2.j  3.+3.j  4.+4.j  5.+5.j  6.+6.j
        7.+7.j]]
y'= [ [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]]
Resultado Global:
y'= [ [  1. +1.j   3. +3.j   5. +5.j   7. +7.j   9. +9.j  11.+11.j
        13.+13.j  15.+15.j]]
```

En este caso apreciamos como el contenido del vector 'y' cambia , de la entrada a la salida. En el caso que quisieramos conservar el valor del vector y, podriamos utilizar otra variable como variable de destino a la salida de la función 'PyPBLAS.pvaxpy':

```
z = PyPBLAS.pvaxpy(alpha,x,y)
```

8.3.5. pvdot

```
dot= PyPBLAS.pvdot(x,y)
```

La función 'PyPBLAS.pvdot' realiza el producto escalar entre el vector y y el vector x, ambos variables tipo PyACTS. La operación se puede resumir como: 'y=xT*y' donde xT representa a la traspuesta de x.

Las características de cada uno de los parámetros son las siguientes:

- Parametros de Entrada
  - 'x': Vector x de tipo PyACTS.
  - 'y': Vector y de tipo PyACTS.
- Parametros de Salida
  - 'dot': Resultado del producto escalar entre x e y. Este valor es una variable global en todos aquellos procesos que han intervenido en el calculo del producto, es decir, que tengan valores de x e y para realizar ese calculo. El proceso PyScaLAPACK.iread==1 siempre tiene este valor por contener datos de los vectores x e y.

Tanto el vector x e y han de ser vectores con elementos de tipo real, que pueden tener precision simple o doble. En el caso de querer realizar el producto escalar de vectores con numeros complejos, se deberá hacer uso de las funciones PyPBLAS.pvdotu (ver sección 8.3.6) y PyPBLAS.pvdotu (ver sección 8.3.7). A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvDOT"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=reshape(array(range(n)),[n,1])
        y=ones([n,1])
        print "x=",transpose(x)
        print "y=",transpose(y)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
#We call PyPblas Routine
dot= PyPBLAS.pvdot(x,y)
if PyACTS.iread==1:
        print "PvDOT=",dot
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvdot.py
Example of using PyPBLAS: PvDOT
Parameters:N= 8 ;Grid: 2 * 2
Block's size: 2 * 2
x= [ [0 1 2 3 4 5 6 7]]
y= [ [1 1 1 1 1 1 1 1]]
PvDOT= 28.0
```

8.3.6. pvdotu

`dotu= PyPBLAS.pvdotu(x,y)`

La función '`PyPBLAS.pvdotu`' realiza el producto escalar entre el vector `y` y el vector `x` ambos vectores PyACTS con elementos complejos. La operación se puede resumir como: '`y=xT*y`' donde `xT` representa a la traspuesta de `x`. Los vectores '`x`' e '`y`' pueden estar o no ditribuidos. En el caso que alguno o ambos no lo estén se distribuirán de forma transparente al usuario. La peculiaridad de esta función con respecto a la función `PyPBLAS.pvdot` () es el tipo de datos que contienen los vectores `x` e `y`. En este caso son de tipo complejo mientras que en la rutina anterior se trataba de números reales. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`x`': Vector x de tipo PyACTS.
  - '`y`': Vector y de tipo PyACTS

- Parametros de Salida
  - 'dotu': Resultado del producto escalar entre x e y. Este valor es una variable global en todos aquellos procesos que han intervenido en el calculo del producto, es decir, que tengan valores de x e y para realizar ese calculo. El proceso PyScaLAPACK.iread==1 siempre tiene este valor por contener datos de los vectores x e y.

Tanto el vector x e y han de ser vectores con elementos de tipo complejo, que pueden tener precision simple o doble. A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvDOTU"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=(1+1j)*ones([n,1])
        y=reshape( (1+1j)*array(range(n),Complex),[n,1])
        print "x'=",transpose(x)
        print "y'=",transpose(y)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
#We call PyPblas Routine
dotu= PyPBLAS.pvdotu(x,y)
if PyACTS.iread==1:
        print "PvDOTU=",dotu
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvdotu.py
Example of using PyPBLAS: PvDOTU
Parameters:N= 8 ;Grid: 2 * 2
Block's size: 2 * 2
x'= [ [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]]
y'= [ [ 0.+0.j  1.+1.j  2.+2.j  3.+3.j  4.+4.j  5.+5.j  6.+6.j
        7.+7.j]]
PvDOTU= 56j
```

8.3.7. pvdotc

```
dotc= PyPBLAS.pvdotc(x,y)
```

La función 'PyPBLAS.pvdotc' realiza el producto escalar entre el vector y y el vector x. La operación se puede resumir como: 'y=xT*y' donde xT representa a la traspuesta de x. Los vectores 'x' e 'y' pueden estar o no ditribuidos. En el caso que alguno o ambos no lo estén se distribuirán de forma transparente al usuario. La peculiaridad de esta función con respecto a la función PyPBLAS.pvdot () es el tipo de datos que contienen los vectores x e y. En este caso son de tipo complejo mientras que en la rutina anterior se trataba de números reales. Las características de cada uno de los parámetros son las siguientes:

- Parametros de Entrada
  - 'x': Vector x de tipo PyACTS.
  - 'y': Vector y de tipo PyACTS.
- Parametros de Salida
  - 'dotc': Resultado del producto escalar entre x e y.
    Este valor es una variable global en todos aquellos procesos que han intervenido en el calculo del producto, es decir, que tengan valores de x e y para realizar ese calculo. El proceso PyScaLAPACK.iread==1 siempre tiene este valor por contener datos de los vectores x e y.

Tanto el vector x e y han de ser vectores con elementos de tipo complejo, que pueden tener precision simple o doble.A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvDOTC"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=(1+1j)*ones([n,1])
        y=reshape( (1+1j)*array(range(n),Complex),[n,1])
        print "x'=",transpose(x)
        print "y'=",transpose(y)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
#We call PyPblas Routine
dotc= PyPBLAS.pvdotc(x,y)
if PyACTS.iread==1:
        print "PvDOTC=",dotc
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvdotc.py
Example of using PyPBLAS: PvDOTC
Parameters:N= 8 ;Grid: 2 * 2
Block's size: 2 * 2
x'= [ [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]]
y'= [ [ 0.+0.j  1.+1.j  2.+2.j  3.+3.j  4.+4.j  5.+5.j  6.+6.j
        7.+7.j]]
PvDOTC= (56+0j)
```

8.3.8. pvnrm2

`nrm2= PyPBLAS.pvnrm2(x)`

La función '`PyPBLAS.pvnrm2`' calcula la norma de un vector `x`. Este vector de entrada `x` deberá ser del tipo PyACTS por lo que habrá que utilizar las funciones que convierten de otros tipos a PyACTS. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`x`': Vector x de la operación descrita.
- `Parametros de Salida`
  - '`nrm2`': Norma del vector `x`.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=4
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvNRM2"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=2*ones([n,1],Float)
        print "x'=",transpose(x)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
#We call PyPblas Routine
nrm2= PyPBLAS.pvnrm2(x)
if PyACTS.iread==1:
        print "PvNRM2=",nrm2
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=4
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvNRM2"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=2*ones([n,1],Float)
        print "x'=",transpose(x)
else:
        x,y=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
#We call PyPblas Routine
nrm2= PyPBLAS.pvnrm2(x)
if PyACTS.iread==1:
        print "PvNRM2=",nrm2
PyACTS.gridexit()
```

8.3.9. pvasum

```
asum= PyPBLAS.pvasum(x)
```

La función '`PyPBLAS.pvasum`' calcula la suma de los valores absolutos del vector `x`, siendo el vector `x` una variable de tipo PyACTS conteniendo un vector en su campo de datos.

- `Parametros de Entrada`
  - '`x`': Vector x de la operación descrita.
- `Parametros de Salida`
  - '`asum`': Suma de los valores absolutos de `x`.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvASUM"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=(2+1j)*ones([n,1])
        print "x'=",transpose(x)
else:
        x=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
#We call PyPblas Routine
asum = PyPBLAS.pvasum(x)
if PyACTS.iread==1:
        print "PvASUM=",asum
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvasum.py
Example of using PyPBLAS: PvASUM
Parameters:N= 8 ;Grid: 2 * 2
Block's size: 2 * 2
x'= [ [ 2.+1.j  2.+1.j  2.+1.j  2.+1.j  2.+1.j  2.+1.j  2.+1.j
        2.+1.j]]
PvASUM= (24+0j)
```

8.3.10. pvamax

`amax,indx= PyPBLAS.pvamax(x)`

La función '`PyPBLAS.pvamax`' obtiene el valor máximo de un vector `x` que se encuentra distribuido entre los procesos de la malla. Esta función devuelve el mayor elemento en valor absoluto del vector `x` y la posición de dicho elemento en el vector. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`x`': Vector x de tipo PyACTS.
- `Parametros de Salida`
  - '`amax`': Valor del elemento con mayor valor absoluto perteneciente a `x`.
  - '`indx`': Posicion del elemento '`amax`'.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *

#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvASUM"
        print "Parameters:N=",n,";Grid:",PyACTS.nprow,"*",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        x=reshape(standard_normal(n),[n,1])
        print "x'=",transpose(x)
else:
        x=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
x=Num2PyACTS(x,ACTS_lib)
#We call PyPblas Routine
amax,indx= PyPBLAS.pvamax(x)
if PyACTS.iread==1:
        print "PvAMAX(",indx,")=",amax
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvamax.py
Example of using PyPBLAS: PvASUM
Parameters:N= 8 ;Grid: 2 * 2
Block's size: 2 * 2
x'= [ [-1.66720867  0.50000608 -1.43065512  0.8494119   0.4929097   0.54123461
       -0.25628906  0.39031151]]
PvAMAX( 0 )= -1.66720867157
```

8.4. PyPBLAS nivel 2

8.4.1. pvgemv

`y= PyPBLAS.pvgemv(alpha,a,x,beta,y[,trans='N'])`

La función '`PyPBLAS.pvgemv`' implementa la siguiente operación entre escalares (`alpha`,`beta`), vectores (`x`, `y`) y matrices (`a`), todos ellos de tipo PyACTS:

$$\alpha \cdot op(A) \cdot x + \beta \cdot y \rightarrow y$$

Donde $op(A)$ depende del valor del parámetro '`trans`':

- `trans='N'`: No se se realiza ninguna operación sobre la matriz.
- `trans='T'`: Se realiza la transpuesta de A: A^T.

- `trans='C'` : Se realiza la transpuesta conjugada de A: A^H.

Las dimensiones de los vectores `x` e `y` y de la matriz `A` han de ser las adecuadas para poder realizar las operaciones correspondientes. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvgemv` informará de ello y no podrá finalizar la operación. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`beta`': Escalar.
  - '`y`': Vector de dimensiones $m \times 1$.
- `Parametros de Salida`
  - '`y`': Vector distribuido donde se almacena el resultado.
  - '`descy`': Descriptor del vector `y`.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvGEMV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha,beta=2,3
        print "alpha=",alpha,"; beta=",beta
        a=identity(n,Float)
        print "a=",a
        x=reshape(array(range(n),Numeric.Float),[n,1])
        print "x'=",transpose(x)
        y=ones([n,1],Float)
        print "y'=",transpose(y)
else:
        a,x,y,alpha,beta=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)

#We call PBLAS routine
y= PyPBLAS.pvgemv(alpha,a,x,beta,y)
y_num=PyACTS2Num(y)
if PyACTS.iread==1:
        print "PvGEMV'=",transpose(y_num)
PyACTS.gridexit()
```

Se ha de observar en este código que hemos El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvgemv.py
Example of using PyPBLAS: PvGEMV
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 2 ; beta= 3
a= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
x'= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
y'= [ [ 1.  1.  1.  1.  1.  1.  1.  1.]]
PvGEMV'= [ [ 3.  3.  3.  3.  3.  3.  3.  3.]]
```

8.4.2. pvhemv

```
y= PyPBLAS.pvhemv(alpha,a,x,beta,y[,uplo='U'])
```

La función '`PyPBLAS.pvhemv`' implementa la siguiente operación entre escalares (`alpha`,`beta`), vectores (`x`, `y`) y matrices (`a`), todos ellos de tipo PyACTS:

$$\alpha \cdot A \cdot x + \beta \cdot y \rightarrow y$$

Las dimensiones de los vectores `x` e `y` y de la matriz `A` han de ser las adecuadas para poder realizar las operaciones correspondientes. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvgemv` informará de ello y no podrá finalizar la operación. Esta rutina se utiliza para el caso que las matrices y vectores sean de tipo complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`beta`': Escalar.
  - '`y`': Vector de dimensiones $m \times 1$.
  - '`uplo`': Caracter que indica el tipo de matriz triangular
    - `uplo='U'` : Triangular Superior (valor por defecto).
    - `uplo='L'` : Triangular Inferior.
- `Parametros de Salida`
  - '`y`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvHEMV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha,beta=2,3
        print "alpha=",alpha,"; beta=",beta
        a=(1+1j)*ones([n,n],Float)
        print "a=",a
        x=(1+2j)*reshape(array(range(n),Numeric.Float),[n,1])
        print "x'=",transpose(x)
        y=1j*ones([n,1],Float)
        print "y'=",transpose(y)
else:
        a,x,y,alpha,beta=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)

#We call PBLAS routine
y= PyPBLAS.pvhemv(alpha,a,x,beta,y)
y_num=PyACTS2Num(y)
if PyACTS.iread==1:
        print "PvHEMV'=",transpose(y_num)
PyACTS.gridexit()
```

Se ha de observar en este código que hemos El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvhemv.py
Example of using PyPBLAS: PvHEMV
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 2 ; beta= 3
a= [[ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j
        1.+1.j]]
x'= [ [ 0. +0.j  1. +2.j  2. +4.j  3. +6.j  4. +8.j  5.+10.j  6.+12.j
        7.+14.j]]
y'= [ [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j
        0.+1.j]]
PvHEMV'= [ [  2.59933934e+11 +6.81574502e+06j   8.88226410e+09 +2.40483881e+07j
         2.37569901e+11 +5.12000098e+03j   8.20592538e+09 +2.40483960e+07j
         0.00000000e+00 +1.00000000e+00j   0.00000000e+00 +1.00000000e+00j
         0.00000000e+00 +1.00000000e+00j   0.00000000e+00 +1.00000000e+00j]]
```

8.4.3. pvsymv

`y= PyPBLAS.pvsymv(alpha,a,x,beta,y)`

La función '`PyPBLAS.pvsymv`' implementa la siguiente operación entre escalares (`alpha`,`beta`), vectores (`x`, `y`) y matrices (`a`), todos ellos de tipo PyACTS:

$$\alpha \cdot A \cdot x + \beta \cdot y \rightarrow y$$

Las dimensiones de los vectores `x` e `y` y de la matriz `A` han de ser las adecuadas para poder realizar las operaciones correspondientes. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLAS.pvgemv` informará de ello y no podrá finalizar la operación. Las características de cada uno de los parámetros ,teniendo en cuenta que son de tipo PyACTS, son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`beta`': Escalar.
  - '`y`': Vector de dimensiones $m \times 1$.
- `Parametros de Salida`

- 'y': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta función:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS: PvSYMV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha,beta=2,3
        print "alpha=",alpha,"; beta=",beta
        a=identity(n,Float)
        print "a=",a
        x=reshape(array(range(n),Float),[n,1])
        print "x'=",transpose(x)
        y=ones([n,1],Float)
        print "y'=",transpose(y)
else:
        a,x,y,alpha,beta=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
#We call PBLAS routine
y= PyPBLAS.pvsymv(alpha,a,x,beta,y)
y_num=PyACTS2Num(y)
if PyACTS.iread==1:
        print "PvSYMV'=alpha*A*x+beta*y=",transpose(y_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
a= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
x'= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
y'= [ [ 1.  1.  1.  1.  1.  1.  1.  1.]]
PvSYMV'=alpha*A*x+beta*y= [ [  3.         -44.99245834  11.         -52.99364471  95.1412353
        69.08868408  52.08119202]]
```

8.4.4. pvtrmv

```
x= PyPBLAS.pvtrmv(a,x,[uplo='U',trans='N',diag='N'])
```

La función '`PyPBLAS.pvtrmv`' implementa la siguiente operación entre una matriz (`a`) y un vector (`x`):

$$A \cdot x \rightarrow x; A^T \cdot x \rightarrow x; A^H \cdot x \rightarrow x$$

Las dimensiones del vector `x` y de la matriz `A` han de ser las adecuadas para poder realizar la multiplicacion entre ambos. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtrmv` informará de ello y no podrá finalizar la operación.

Por otro lado es importante señalar que esta rutina se provee únicamente para matrices triangulares (por esta razón el nombre de la función contiene 'tr'). En el caso que se pasara una matriz no triangular como matriz '`a`', la rutina `PyBLAS_trmv` realizará los cálculos como si de una triangular se tratara. Es decir, supondrá que los elementos por debajo de la diagonal (o por encima, dependiendo del valor de '`diag`') son ceros.

Los parámetros de entrada deberán ser de tipo real, no realizándose el cálculo en el caso que alguno de ellos sea de tipo complejo. Las características de cada uno de los parámetros ,teniendo en cuenta que son de tipo PyACTS, son las siguientes:

- `Parametros de Entrada`
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`uplo`': Caracter que indica el tipo de matriz triangular
    - `uplo='U'` : Triangular Superior (valor por defecto).
    - `uplo='L'` : Triangular Inferior.
  - '`trans`': Caracter que indica si se debe realizar la transpuesta de la matriz '`a`'.
    - `diag='T'` : Matriz transpuesta.
    - `diag='N'` : Matriz NO transpuesta (valor por defecto).
  - '`diag`': Caracter que indica si la matriz triangular es unitaria o no:
    - `diag='N'` : Matriz triangular NO unitaria (valor por defecto).
    - `diag='U'` : Matriz triangular Unitaria.
- `Parametros de Salida`
  - '`x`': Vector distribuido donde se almacena el resultado.

Debemos señalar en este punto que los tipos de datos de los vectores y matrices son únicamente reales, de esto se deduce que no se puede dar el caso de indicar `trans='H'` puesto que la conjugada de un número real es el mismo número real.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvTRMV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([n,n],Float)
        print "a=",a
        x=reshape(array(range(n),Float),[n,1])
        print "x'=",transpose(x)
else:
        a,x,alpha=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
#We call PBLAS routine
x= PyPBLAS.pvtrmv(a,x)
x_num=PyACTS2Num(x)
if PyACTS.iread==1:
        print "PvTRMV'=",transpose(x_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtrmv.py
Example of using PyPBLAS 2: PvTRMV
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]]
x'= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
PvTRMV'= [ [ 28.  28.  27.  25.  22.  18.  13.   7.]]
```

En el ejemplo mostrado podemos comprobar cómo se ha realizado la operación descrita con los elementos que se encuentran en la diagonal y por encima de ella. Aquellos elementos situados por debajo de la diagonal se suponen igual a cero. Si consideraramos que la matriz `a` fuera diagonal inferior, habría que indicarlo del siguiente modo:

```
x= PyPBLAS.pvtrmv(a,x,uplo='L')
```

Y entonces, el resultado que obtendríamos sería el siguiente:

```
PvTRMV'= [ [  0.   1.   3.   6.  10.  15.  21.  28.]]
```

8.4.5. pvtrsv

```
x= PyPBLAS.pvtrsv(a,x,[uplo='U',trans='N',diag='N'])
```

La función '`PyPBLAS.pvtrsv`' resuleve un sistema de ecuaciones del siguiente tipo:

$$A^{-1} \cdot x \rightarrow x; A^{-T} \cdot x \rightarrow x; A^{-H} \cdot x \rightarrow x$$

Las dimensiones del vector `x` y de la matriz `A` han de ser las adecuadas para poder realizar la multiplicacion entre ambos. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtrsv` informará de ello y no podrá finalizar la operación.

Por otro lado es importante señalar que esta rutina se provee únicamente paramatrices triangulares (por esta razón el nombre de la función contiene 'tr'). En el caso que se pasara una matriz no triangular como matriz '`a`', la rutina `PyBLAS_trsv` realizará los cálculos como si de una triangular se tratara. Es decir, supondrá que los elementos por debajo de la diagonal (o por encima, dependiendo del valor de '`diag`') son ceros.

Los parámetros de entrada pueden ser de tipo real o complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`uplo`': Caracter que indica el tipo de matriz triangular
    - `uplo='U'`: Triangular Superior (valor por defecto).
    - `uplo='L'`: Triangular Inferior.
  - '`trans`': Caracter que indica si se debe realizar la transpuesta de la matriz '`a`'.
    - `diag='T'`: Matriz transpuesta.
    - `diag='N'`: Matriz NO transpuesta (valor por defecto).
  - '`diag`': Caracter que indica si la matriz triangular es unitaria o no:
    - `diag='N'`: Matriz triangular NO unitaria (valor por defecto).
    - `diag='U'`: Matriz triangular Unitaria.
- `Parametros de Salida`
  - '`x`': Vector distribuido donde se almacena el resultado.

Debemos señalar en este punto que los tipos de datos de los vectores y matrices son únicamente reales, de esto se deduce que no se puede dar el caso de indicar `trans='H'` puesto que la conjugada de un número real es el mismo número real.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvTRSV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=4*identity(n,Float)
        print "a=",a
        x=reshape(array(range(n),Float),[n,1])
        print "x'=",transpose(x)
else:
        a,x,alpha=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
#We call PBLAS routine
x= PyPBLAS.pvtrsv(a,x)
x_num=PyACTS2Num(x)
if PyACTS.iread==1:
        print "PvTRSV'=",transpose(x_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtrsv.py
Example of using PyPBLAS 2: PvTRSV
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 4.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  4.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  4.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  4.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  4.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  4.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  4.]]
x'= [ [ 0.  1.  2.  3.  4.  5.  6.  7.]]
PvTRSV'= [ [ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75]]
```

8.4.6. pvger

`a= PyPBLAS.pvger(alpha,x,y,a)`

La función '`PyPBLAS.pvger`' implementa la siguiente operación:

$$\alpha \cdot x \times y^T + A \rightarrow A$$

Las dimensiones de los vectores `x` e `y` y de la matriz `A` han

de ser las adecuadas para poder realizar la multiplicacion entre ambos. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvger` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo real, si los elementos fueran complejos se deberá utilizar la rutina `PyPBLAS.pvgeru` 8.4.7 o `PyPBLAS.pvgerc` 8.4.8. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - 'alpha':Escalar.
  - 'x': Vector de dimensiones $m \times 1$.
  - 'y': Vector de dimensiones $m \times 1$.
  - 'a': Matriz de dimensiones $m \times n$.
- `Parametros de Salida`
  - 'a': Matriz distribuido donde se almacena el resultado.

Tanto los vectores y las matrices de entrada han de ser de tipo PyACTS. El resultado es un vector de tipo PyACTs también. A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvGER"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=ones([n,1])
        print "x'=",transpose(x)
        y=reshape(array(range(n)),[n,1])
        print "y'=",transpose(y)
        a=identity(n)
        print "a=",a
else:
        alpha,a,x,y=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvger(alpha,x,y,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvGER=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvger.py
Example of using PyPBLAS 2: PvGER
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [1 1 1 1 1 1 1 1]]
y'= [ [0 1 2 3 4 5 6 7]]
a= [[1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1]]
PvGER= [[  1.   0.   0.   0.   0.   0.   0.   0.]
 [  5.   6.   5.   5.   5.   5.   5.   5.]
 [ 10.  10.  11.  10.  10.  10.  10.  10.]
 [ 15.  15.  15.  16.  15.  15.  15.  15.]
 [ 20.  20.  20.  20.  21.  20.  20.  20.]
 [ 25.  25.  25.  25.  25.  26.  25.  25.]
 [ 30.  30.  30.  30.  30.  30.  31.  30.]
 [ 35.  35.  35.  35.  35.  35.  35.  36.]]
```

8.4.7. pvgeru

`a= PyPBLAS.pvgeru(alpha,x,y,a)`

La función '`PyPBLAS.pvgeru`' implementa la siguiente operación entre vectores y matrices con elementos de tipo complejo:

$$\alpha \cdot x \times y^T + A \rightarrow A$$

Las dimensiones de los vectores `x` e `y` y de la matriz `A` han de ser las adecuadas para poder realizar la multiplicacion entre ambos. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvgeru` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`y`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
- `Parametros de Salida`
  - '`a`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvGERU"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=ones([n,1])+1.j*reshape(range(n),[n,1])
        print "x'=",transpose(x)
        y=reshape(range(n),[n,1])+1.j*ones([n,1])
        print "y'=",transpose(y)
        a=identity(n,Complex)
        print "a=",a
else:
        alpha,a,x,y=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvgeru(alpha,x,y,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvGERU=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvgeru.py
Example of using PyPBLAS 2: PvGERU
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [ 1.+0.j  1.+1.j  1.+2.j  1.+3.j  1.+4.j  1.+5.j]]
y'= [ [ 0.+1.j  1.+1.j  2.+1.j  3.+1.j  4.+1.j  5.+1.j]]
a= [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
PvGERU= [[  1.  +5.j  -5.  +5.j -10.  +5.j -15.  +5.j -20.  +5.j -25.  +5.j]
 [  5.  +5.j   1. +10.j  -5. +15.j -10. +20.j -15. +25.j -20. +30.j]
 [ 10.  +5.j   5. +15.j   1. +25.j  -5. +35.j -10. +45.j -15. +55.j]
 [ 15.  +5.j  10. +20.j   5. +35.j   1. +50.j  -5. +65.j -10. +80.j]
 [ 20.  +5.j  15. +25.j  10. +45.j   5. +65.j   1. +85.j  -5.+105.j]
 [ 25.  +5.j  20. +30.j  15. +55.j  10. +80.j   5.+105.j   1.+130.j]]
```

8.4.8. pvgerc

`a= PyPBLAS.pvgerc(alpha,x,y,a)`

La función '`PyPBLAS.pvgerc`' implementa la siguiente operación entre vectores y matrices con elementos de tipo complejo:

$$\alpha \cdot x \times y^H + A \rightarrow A$$

Se ha de tener en cuenta que y^H será el vector transpuesto conjugado de `y`, mientras que y^T se corresponde con la transpuesta de `y` implementado en la función anterior `PyBLAS_pvgeru` 8.4.8. Las dimensiones de los vectores `x` e `y` y de la matriz `A` han de ser las adecuadas para poder realizar la multiplicacion entre ambos. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvgerc` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`y`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
- `Parametros de Salida`
  - '`a`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvGERC"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=ones([n,1])+1.j*reshape(range(n),[n,1])
        print "x'=",transpose(x)
        y=reshape(range(n),[n,1])+1.j*ones([n,1])
        print "y'=",transpose(y)
        a=identity(n,Complex)
        print "a=",a
else:
        alpha,a,x,y=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvgerc(alpha,x,y,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvGERC=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvgerc.py
Example of using PyPBLAS 2: PvGERC
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [ 1.+0.j  1.+1.j  1.+2.j  1.+3.j  1.+4.j  1.+5.j]]
y'= [ [ 0.+1.j  1.+1.j  2.+1.j  3.+1.j  4.+1.j  5.+1.j]]
a= [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
PvGERC= [[  1.  -5.j   5.  -5.j  10.  -5.j  15.  -5.j  20.  -5.j  25.  -5.j]
 [  5.  -5.j  11.  +0.j  15.  +5.j  20. +10.j  25. +15.j  30. +20.j]
 [ 10.  -5.j  15.  +5.j  21. +15.j  25. +25.j  30. +35.j  35. +45.j]
 [ 15.  -5.j  20. +10.j  25. +25.j  31. +40.j  35. +55.j  40. +70.j]
 [ 20.  -5.j  25. +15.j  30. +35.j  35. +55.j  41. +75.j  45. +95.j]
 [ 25.  -5.j  30. +20.j  35. +45.j  40. +70.j  45. +95.j  51.+120.j]]
```

8.4.9. pvher

`a= PyPBLAS.pvher(alpha,x,a,[uplo='U'])`

La función '`PyPBLAS.pvher`' implementa la siguiente operación entre vectores y matrices con elementos de tipo complejo:

$$\alpha \cdot x \times x^H + A \rightarrow A$$

Se ha de tener en cuenta que x^H será el vector transpuesto conjugado de `x`, mientras que x^T se corresponde con la transpuesta de `x`. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvher` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`uplo`': Caracter que indica cómo queremos obtener el resultado en '`a`'.
    - `uplo='U'`: Triangular Superior (valor por defecto).
    - `uplo='L'`: Triangular Inferior.
- `Parametros de Salida`
  - '`a`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvHER"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=ones([n,1])+1.j*reshape(range(n),[n,1])
        print "x'=",transpose(x)
        a=identity(n,Complex)
        print "a=",a
else:
        alpha,a,x=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvher(alpha,x,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvHER=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvher.py
Example of using PyPBLAS 2: PvHER
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [ 1.+0.j  1.+1.j  1.+2.j  1.+3.j  1.+4.j  1.+5.j]]
a= [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
PvHER= [[   6. +0.j    0. +0.j    0. +0.j    0. +0.j    0. +0.j    0. +0.j]
 [   5. -5.j   11. +0.j    0. +0.j    0. +0.j    0. +0.j    0. +0.j]
 [   5.-10.j   15. -5.j   26. +0.j    0. +0.j    0. +0.j    0. +0.j]
 [   5.-15.j   20.-10.j   35. -5.j   51. +0.j    0. +0.j    0. +0.j]
 [   5.-20.j   25.-15.j   45.-10.j   65. -5.j   86. +0.j    0. +0.j]
 [   5.-25.j   30.-20.j   55.-15.j   80.-10.j  105. -5.j  131. +0.j]]
```

Se ha de tener en cuenta que el parámetro por defecto para `uplo='U'`, es decir, realizaremos los calculos unicamente que se encuentre en y por encima de la diagonal de la matriz resultante. Si quisieramos obtener la parte por debajo de

la diagonal llamaríamos a la rutina del siguiente modo:

```
a= PyPBLAS.pvher(alpha,x,a,uplo='L')
```

8.4.10. pvher2

```
a= PyPBLAS.pvher2(alpha,x,y,a,[uplo='U'])
```

La función '`PyPBLAS.pvher2`' implementa la siguiente operación entre vectores y matrices con elementos de tipo complejo:

$$\alpha \cdot x \times y^H + y(alpha \cdot x)^H + A \rightarrow A$$

Se ha de tener en cuenta que x^H será el vector transpuesto conjugado de `x`, mientras que x^T se corresponde con la transpuesta de `x`. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvher2` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`y`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`uplo`': Caracter que indica cómo queremos obtener el resultado en '`a`'.
    - `uplo='U'`: Triangular Superior (valor por defecto).
    - `uplo='L'`: Triangular Inferior.
- `Parametros de Salida`
  - '`a`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvHER2"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=ones([n,1])+1.j*reshape(range(n),[n,1])
        print "x'=",transpose(x)
        y=reshape(range(n),[n,1])+1.j*ones([n,1])
        print "y'=",transpose(y)
        a=identity(n,Complex)
        print "a=",a
else:
        alpha,a,x,y=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvher2(alpha,x,y,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvHER2=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvher2.py
Example of using PyPBLAS 2: PvHER2
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [ 1.+0.j  1.+1.j  1.+2.j  1.+3.j  1.+4.j  1.+5.j]]
y'= [ [ 0.+1.j  1.+1.j  2.+1.j  3.+1.j  4.+1.j  5.+1.j]]
a= [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
PvHER2= [[   1.+0.j    0.+0.j    0.+0.j    0.+0.j    0.+0.j    0.+0.j]
 [  10.+0.j   21.+0.j    0.+0.j    0.+0.j    0.+0.j    0.+0.j]
 [  20.+0.j   30.+0.j   41.+0.j    0.+0.j    0.+0.j    0.+0.j]
 [  30.+0.j   40.+0.j   50.+0.j   61.+0.j    0.+0.j    0.+0.j]
 [  40.+0.j   50.+0.j   60.+0.j   70.+0.j   81.+0.j    0.+0.j]
 [  50.+0.j   60.+0.j   70.+0.j   80.+0.j   90.+0.j  101.+0.j]]
```

Se ha de tener en cuenta que el parámetro por defecto para `uplo='U'`, es decir, realizaremos los calculos unicamente que se encuentre en y por encima de la diagonal de la matriz resultante. Si quisieramos obtener la parte por debajo de la diagonal llamaríamos a la rutina del siguiente modo:

```
a,desca= PyPBLAS.pvher2(alpha,x,y,a,uplo='L')
```

8.4.11. pvsyr

```
a= PyPBLAS.pvsyr(alpha,x,a,[uplo='U'])
```

La función '`PyPBLAS.pvsyr`' implementa la siguiente operación entre vectores y matrices con elementos de tipo real:

$$\alpha \cdot x \times x^T + A \rightarrow A$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvsyr` informará de ello y no podrá finalizar la operación.

Por otro lado, debido a las características de la operacion a realizar la matriz resultante es simétrica, por tanto únicamente se proporcionará el resultado de los elementos situados en la diagonal principal y por encima o por debajo de la misma, dependiendo del parametro '`uplo`'. Esta rutina se provee únicamente para matrices con elementos de tipo complejo.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`uplo`': Caracter que indica cómo queremos obtener el resultado en '`a`'.
    - `uplo='U'`: Triangular Superior (valor por defecto).

    - uplo='L': Triangular Inferior.

- Parametros de Salida
  - 'a': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvSYR"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=reshape(range(n),[n,1])
        print "x'=",transpose(x)
        a=identity(n)
        print "a=",a
else:
        alpha,a,x=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvsyr(alpha,x,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvSYR=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvsyr.py
Example of using PyPBLAS 2: PvSYR
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [0 1 2 3 4 5 6 7]]
a= [[1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1]]
PvSYR= [[   1.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    6.    0.    0.    0.    0.    0.    0.]
 [   0.   10.   21.    0.    0.    0.    0.    0.]
 [   0.   15.   30.   46.    0.    0.    0.    0.]
 [   0.   20.   40.   60.   81.    0.    0.    0.]
 [   0.   25.   50.   75.  100.  126.    0.    0.]
 [   0.   30.   60.   90.  120.  150.  181.    0.]
 [   0.   35.   70.  105.  140.  175.  210.  246.]]
```

Se ha de tener en cuenta que el parámetro por defecto para `uplo='U'`, es decir, realizaremos los calculos unicamente que se encuentre en y por encima de la diagonal de la matriz resultante. Si quisieramos obtener la parte por debajo de la diagonal llamaríamos a la rutina del siguiente modo:

```
a,desca= PyPBLAS.pvsyr(alpha,x,a,uplo='L')
```

8.4.12. pvsyr2

```
a= PyPBLAS.pvsyr2(alpha,x,y,a,[uplo='U'])
```

La función '`PyPBLAS.pvsyr2`' implementa la siguiente operación entre vectores y matrices con elementos de tipo real:

$$\alpha \cdot x \times y^T + y(\alpha \cdot x)^T + A \rightarrow A$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvsyr2` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`x`': Vector de dimensiones $m \times 1$.
  - '`y`': Vector de dimensiones $m \times 1$.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`uplo`': Caracter que indica cómo queremos obtener el resultado en '`a`'.

    - uplo='U': Triangular Superior (valor por defecto).
    - uplo='L': Triangular Inferior.

- Parametros de Salida
    - 'a': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 2: PvSYR2"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        alpha=5
        print "alpha=",alpha
        x=reshape(range(n),[n,1])
        print "x'=",transpose(x)
        y=ones([n,1])
        print "y'=",transpose(y)
        a=identity(n,Float)
        print "a=",a
else:
        alpha,a,x,y=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
y=Num2PyACTS(y,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvsyr2(alpha,x,y,a)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvSYR2'=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvsyr2.py
Example of using PyPBLAS 2: PvSYR2
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
alpha= 5
x'= [ [0 1 2 3 4 5 6 7]]
y'= [ [1 1 1 1 1 1 1 1]]
a= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
PvSYR2= [[  1.   0.   0.   0.   0.   0.   0.   0.]
 [  5.  11.   0.   0.   0.   0.   0.   0.]
 [ 10.  15.  21.   0.   0.   0.   0.   0.]
 [ 15.  20.  25.  31.   0.   0.   0.   0.]
 [ 20.  25.  30.  35.  41.   0.   0.   0.]
 [ 25.  30.  35.  40.  45.  51.   0.   0.]
 [ 30.  35.  40.  45.  50.  55.  61.   0.]
 [ 35.  40.  45.  50.  55.  60.  65.  71.]]
```

Se ha de tener en cuenta que el parámetro por defecto para `uplo='U'`, es

decir, realizaremos los calculos unicamente que se encuentre en y por encima de la diagonal de la matriz resultante. Si quisieramos obtener la parte por debajo de la diagonal llamaríamos a la rutina del siguiente modo:

```
a,desca= PyPBLAS.pvsyr2(alpha,x,y,a,uplo='L')
```

8.5. PyPBLAS nivel 3

8.5.1. pvgemm

```
c= PyPBLAS.pvgemm(alpha,a,b,beta,c,[transa='N',transb='N'])
```

La función '`PyPBLAS.pvgemm`' implementa la siguiente operación entre matrices:

$$\alpha \cdot op(A) \times op(B) + \beta \cdot C \rightarrow C$$

Donde $op(A)$ depende del valor del parámetro '`trans`':

- `trans='N'`: No se se realiza ninguna operación sobre la matriz.
- `trans='T'`: Se realiza la transpuesta de A: A^T.
- `trans='C'`: Se realiza la transpuesta conjugada de A: A^H.

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvgemm` informará de ello y no podrá finalizar la operación.

Esta rutina se provee únicamente para matrices con elementos de tipo complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - 'alpha':Escalar.
  - 'a': Matriz de dimensiones $m \times k$.
  - 'b': Matriz de dimensiones $k \times n$.
  - 'beta':Escalar.
  - 'c': Matriz de dimensiones $m \times n$.
  - 'transa': Caracter que indica la opearación a realizar sobre la matriz a:
    - uplo='N':No transpuesta, es decir, no se realiza operación.(valor por defecto)
    - uplo='T':Transpuesta.
    - uplo='H': Transpuesta conjugada.
- `Parametros de Salida`
  - 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
m,n,k=8,8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvGEMM"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*k),[m,k])
        print "a=",a
        b=reshape(range(k*n),[k,n])
        print "b=",b
        c=reshape(range(m*n),[m,n])
        print "c=",c
        alpha=2.
        beta=3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,b,beta,c=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
a= PyPBLAS.pvgemm(alpha,a,b,beta,c)
a_num=PyACTS2Num(a)
if PyACTS.iread==1:
        print "PvGEMM=",transpose(a_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvgemm.py
Example of using PyPBLAS 3: PvGEMM
N= 8 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]
 [36 37 38 39 40 41]
 [42 43 44 45 46 47]]
b= [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
c= [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]
 [48 49 50 51 52 53 54 55]
 [56 57 58 59 60 61 62 63]]
alpha= 2.0 ; beta= 3.0
PvGEMM= [[   880.   2344.   3808.   5272.   6736.   8200.   9664.  11128.]
 [   913.   2449.   3985.   5521.   7057.   8593.  10129.  11665.]
 [   946.   2554.   4162.   5770.   7378.   8986.  10594.  12202.]
 [   979.   2659.   4339.   6019.   7699.   9379.  11059.  12739.]
 [  1012.   2764.   4516.   6268.   8020.   9772.  11524.  13276.]
 [  1045.   2869.   4693.   6517.   8341.  10165.  11989.  13813.]
 [  1078.   2974.   4870.   6766.   8662.  10558.  12454.  14350.]
 [  1111.   3079.   5047.   7015.   8983.  10951.  12919.  14887.]]
```

Los paramentros `transa`, y `transb` indican la operación a realizar sobre las matrices `a` y `b`, respectivamente tal y como se ha detallado anteriormente. Estos parámetros son opcionales, el valor por defector es '`N`', es decir, no realiza ninguna operación sobre la matriz correspondiente. En el caso que, por ejemplo quisieramos realizar el cálculo con la transpuesta de `b` la forma de llamar a la rutina sería:

```
c=PyPBLAS.pvgemm(alpha,a,b,beta,c,transb='T')
```

8.5.2. pvsymm

```
c=PyPBLAS.pvsymm(alpha,a,b,beta,c[,side='L',uplo='U'])
```

La función '`PyPBLAS.pvsymm`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices son simétricas:

$$\alpha \cdot A \times B + \beta \cdot C \rightarrow C; \alpha \cdot B \times A + \beta \cdot C \rightarrow C$$

donde

$$A = A^T$$

. En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `BLAS_pvsymm` informará de ello y no podrá finalizar la operación.

Esta rutina se provee tanto para matrices con elementos de tipo real o de tipo complejo. Los vectores y matrices de entrada tendrán que ser elementos del tipo PyACTS. Las características de cada uno de los parámetros, siendo todos de tipo PyACTS, son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times m$, que cumple $A = A^T$.
  - '`b`': Matriz de dimensiones $k \times n$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times n$.
  - '`side`': Caracter que indica la posición de la matriz `a` en la operación:
    - `side='L'` : (Valor por defecto).La operación que se realiza es $\alpha \cdot A \times B + \beta \cdot C$.
    - `side='R'` : La operación que se realiza es $\alpha \cdot B \times A + \beta \cdot C$.
  - '`uplo`': Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
    - `uplo='U'` : (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
    - `uplo='L'` : Se obtienen los resultados en la diagonal principal y por debajo de ella.
- `Parametros de Salida`
  - '`c`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
m,n=6,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvSYMM"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*m),[m,m])
        print "a=",a
        b=reshape(range(m*n),[m,n])
        print "b=",b
        c=reshape(range(m*n),[m,n])
        print "c=",c
        alpha=2.
        beta=3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,b,beta,c=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvsymm(alpha,a,b,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvSYMM=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvsymm.py
Example of using PyPBLAS 3: PvSYMM
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
b= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
c= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
alpha= 2.0 ; beta= 3.0
PvSYMM= [[  660.  1758.  2796.  3714.  4452.  4950.]
 [  693.  1853.  2943.  3903.  4673.  5193.]
 [  726.  1948.  3090.  4092.  4894.  5436.]
 [  759.  2043.  3237.  4281.  5115.  5679.]
 [  792.  2138.  3384.  4470.  5336.  5922.]
 [  825.  2233.  3531.  4659.  5557.  6165.]]
```

Los paramentros `side`, y `uplo` son parámetros opcionales que tienen un valor establecido por defecto. En el caso que no se especifiquen estos parámetros tomaraán sus valores por defecto. Para especificar un valor diferente se puede realizar del siguiente modo:

```
c=PyPBLAS.pvsymm(alpha,a,b,beta,c,side='R',uplo='L')
```

8.5.3. pvhemm

`c=PyPBLAS.pvhemm(alpha,a,b,beta,c[,side='L',uplo='U'])`

La función '`PyPBLAS.pvhemm`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices han de ser con elementos de tipo complejo:

$$\alpha \cdot A \times B + \beta \cdot C \rightarrow C; \alpha \cdot B \times A + \beta \cdot C \rightarrow C$$

donde

$$A = A^H$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvhemm` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices con elementos de tipo complejo pero de carácter simetrétrico, es decir tomara las mátrices como simétricas desechando uno de los lados de la diagonal principal. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - ‘`alpha`’:Escalar.
  - ‘`a`’: Matriz de dimensiones $m \times m$ que cumple $A = A^H$.
  - ‘`b`’: Matriz de dimensiones $k \times n$.
  - ‘`beta`’:Escalar.
  - ‘`c`’: Matriz de dimensiones $m \times n$.
  - ‘`side`’: Caracter que indica la posición de la matriz `a` en la operación:
    - `side='L'` : (Valor por defecto).La operación que se realiza es $\alpha \cdot A \times B + \beta \cdot C$.
    - `side='R'` : La operación que se realiza es $\alpha \cdot B \times A + \beta \cdot C$.
  - ‘`uplo`’: Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
    - `uplo='U'` : (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
    - `uplo='L'` : Se obtienen los resultados en la diagonal principal y por debajo de ella.
- `Parametros de Salida`
  - ‘`c`’: Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
m,n=6,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvHEMM"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([m,m])+1.j*reshape(range(m*m),[m,m])
        print "a=",a
        b=ones([m,m])-1.j*reshape(range(m*n),[m,n])
        print "b=",b
        c=1.j*ones([m,n])
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,b,beta,c=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvsymm(alpha,a,b,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvHEMM=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvhemm.py
Example of using PyPBLAS 3: PvSYMM
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 1. +0.j  1. +1.j  1. +2.j  1. +3.j  1. +4.j  1. +5.j]
 [ 1. +6.j  1. +7.j  1. +8.j  1. +9.j  1.+10.j  1.+11.j]
 [ 1.+12.j  1.+13.j  1.+14.j  1.+15.j  1.+16.j  1.+17.j]
 [ 1.+18.j  1.+19.j  1.+20.j  1.+21.j  1.+22.j  1.+23.j]
 [ 1.+24.j  1.+25.j  1.+26.j  1.+27.j  1.+28.j  1.+29.j]
 [ 1.+30.j  1.+31.j  1.+32.j  1.+33.j  1.+34.j  1.+35.j]]
b= [[ 1. +0.j  1. -1.j  1. -2.j  1. -3.j  1. -4.j  1. -5.j]
 [ 1. -6.j  1. -7.j  1. -8.j  1. -9.j  1.-10.j  1.-11.j]
 [ 1.-12.j  1.-13.j  1.-14.j  1.-15.j  1.-16.j  1.-17.j]
 [ 1.-18.j  1.-19.j  1.-20.j  1.-21.j  1.-22.j  1.-23.j]
 [ 1.-24.j  1.-25.j  1.-26.j  1.-27.j  1.-28.j  1.-29.j]
 [ 1.-30.j  1.-31.j  1.-32.j  1.-33.j  1.-34.j  1.-35.j]]
c= [[ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]
 [ 0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j  0.+1.j]]
alpha= 2.0 ; beta= 3.0
PvSYMM= [[  672.-147.j  1752. -85.j  2772. -33.j  3672.  +9.j  4392. +41.j
        4872. +63.j]
 [  702.-159.j  1844. -97.j  2916. -45.j  3858.  -3.j  4610. +29.j
        5112. +51.j]
 [  732.-171.j  1936.-109.j  3060. -57.j  4044. -15.j  4828. +17.j
        5352. +39.j]
 [  762.-183.j  2028.-121.j  3204. -69.j  4230. -27.j  5046.  +5.j
        5592. +27.j]
 [  792.-195.j  2120.-133.j  3348. -81.j  4416. -39.j  5264.  -7.j
        5832. +15.j]
 [  822.-207.j  2212.-145.j  3492. -93.j  4602. -51.j  5482. -19.j
        6072.  +3.j]]
```

Los paramentros `side` y `uplo` son parámetros opcionales que tienen un valor establecido por defecto. En el caso que no se especifiquen estos parámetros tomaraán sus valores por defecto. Para especificar un valor diferente se puede realizar del siguiente modo:

```
c,descc=PyPBLAS.pvsymm(alpha,a,b,beta,c,side='R',uplo='L')
```

8.5.4. pvsyrk

`c=PyPBLAS.pvsyrk(alpha,a,beta,c[,uplo,trans])`

La función '`PyPBLAS.pvsyrk`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices son simétricas:

$$\alpha \cdot A \times A^T + \beta \cdot C \rightarrow C; \alpha \cdot A^T \times A + \beta \cdot C \rightarrow C;$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvsyrk` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices simétricas con elementos de tipo real o complejo. Un detalle importante en esta rutina es la suposición que las matrices de entrada son simétricas, de este modo se realizan los cálculos con la parte superior (o inferior) a la diagonal. El resultado válido en la matriz devuelta por la función será el situado en la posición que le indiquemos a través del parámetro `uplo`. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times m$, que cumple $A = A^T$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times m$.
  - '`uplo`': Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
    - `uplo='U'` : (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
    - `uplo='L'` : Se obtienen los resultados en la diagonal principal y por debajo de ella.
  - '`trans`': A partir de este parámtetro se pueden realizar dos operaciones diferentes, realizando (o no) la transpuesta a la matriz `a`:
    - `trans='N'` : (Valor por defecto).No se realiza la transpuesta y la operación que se lleva a cabo es : $\alpha \cdot A \times A^T + \beta \cdot C \rightarrow C$.
    - `trans='T'` : Se realiza la transpuesta y la operación que se lleva a cabo es : $\alpha \cdot A^T \times A + \beta \cdot C \rightarrow C$.
- `Parametros de Salida`
  - '`c`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n,k=6,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvSYRK"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([n,k])
        print "a=",a
        c=reshape(range(n*n),[n,n])
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,beta,c=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvsyrk(alpha,a,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvSYRK=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvsyrk.py
Example of using PyPBLAS 3: PvSYRK
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
c= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
alpha= 2.0 ; beta= 3.0
PvSYRK= [[  12.    6.   12.   18.   24.   30.]
 [  15.   33.   13.   19.   25.   31.]
 [  18.   36.   54.   20.   26.   32.]
 [  21.   39.   57.   75.   27.   33.]
 [  24.   42.   60.   78.   96.   34.]
 [  27.   45.   63.   81.   99.  117.]]
```

Podemos ver en este ejemplo, que el resultado no es una matriz simétrica aunque debiera serlo puesto que `a` y `c` lo son y el resultado de la operación descrita ha de genera una matriz simétrica. Como por defecto `uplo='U'`, el resultado es una matriz simétrica con los valores correctos situados en y por encima de la diagonal. Si ejecutaramos `c,descc=PyPBLAS.pvsyrk(alpha,a,beta,c,uplo='L')`, podríamos comprobar que los valores correctos se obtienen en y por debajo de la diagonal principal.

8.5.5. pvherk

`c=PyPBLAS.pvherk(alpha,a,beta,c[,uplo,trans])`

La función '`PyPBLAS.pvherk`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices son simétricas y con elementos de tipo complejo:

$$\alpha \cdot A \times A^H + \beta \cdot C \rightarrow C; \alpha \cdot A^H \times A + \beta \cdot C \rightarrow C;$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvherk` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices simétricas con elementos de tipo complejo. Un detalle importante en esta rutina es la suposición que las matrices de entrada son simétricas, de este modo se realizan los cálculos con la parte superior (o inferior) a la diagonal. El resultado válido en la matriz devuelta por la función será el situado en la posición que le indiquemos a través del parámetro `uplo`. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times m$, que cumple $A = A^T$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times m$.
  - '`uplo`': Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:

- uplo='U': (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
- uplo='L': Se obtienen los resultados en la diagonal principal y por debajo de ella.

- 'trans': A partir de este parámtetro se pueden realizar dos operaciones diferentes, realizando (o no) la transpuesta a la matriz a:
  - trans='N': (Valor por defecto).No se realiza la transpuesta y la operación que se lleva a cabo es : $\alpha \cdot A \times A^H + \beta \cdot C \rightarrow C$.
  - trans='H': Se realiza la transpuesta y la operación que se lleva a cabo es : $\alpha \cdot A^H \times A + \beta \cdot C \rightarrow C$.

- Parametros de Salida
  - 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
n,k=6,6
def make_sym(x,y):
        return x*y+(x+y)*1j
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvHERK"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([n,k])+1j*ones([n,k])
        print "a=",a
        c=fromfunction(make_sym,(n,n))
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,beta,c=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvherk(alpha,a,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvHERK=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvherk.py
Example of using PyPBLAS 3: PvHERK
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]]
c= [[  0. +0.j   0. +1.j   0. +2.j   0. +3.j   0. +4.j   0. +5.j]
 [  0. +1.j   1. +2.j   2. +3.j   3. +4.j   4. +5.j   5. +6.j]
 [  0. +2.j   2. +3.j   4. +4.j   6. +5.j   8. +6.j  10. +7.j]
 [  0. +3.j   3. +4.j   6. +5.j   9. +6.j  12. +7.j  15. +8.j]
 [  0. +4.j   4. +5.j   8. +6.j  12. +7.j  16. +8.j  20. +9.j]
 [  0. +5.j   5. +6.j  10. +7.j  15. +8.j  20. +9.j  25.+10.j]]
alpha= 2.0 ; beta= 3.0
PvHERK= [[ 24. +0.j   0. +1.j   0. +2.j   0. +3.j   0. +4.j   0. +5.j]
 [ 24. +3.j  27. +0.j   2. +3.j   3. +4.j   4. +5.j   5. +6.j]
 [ 24. +6.j  30. +9.j  36. +0.j   6. +5.j   8. +6.j  10. +7.j]
 [ 24. +9.j  33.+12.j  42.+15.j  51. +0.j  12. +7.j  15. +8.j]
 [ 24.+12.j  36.+15.j  48.+18.j  60.+21.j  72. +0.j  20. +9.j]
 [ 24.+15.j  39.+18.j  54.+21.j  69.+24.j  84.+27.j  99. +0.j]]
```

Podemos ver en este ejemplo, que el resultado no es una matriz simétrica aunque debiera serlo puesto que `a` y `c` lo son y el resultado de la operación descrita ha de genera una matriz simétrica. Como por defecto `uplo='U'`, el resultado es una matriz simétrica con los valores correctos situados en y por encima de la diagonal. Si ejecutaramos `c=PyPBLAS.pvherk(alpha,a,beta,c,uplo='L')`, podríamos comprobar que los valores correctos se obtienen en y por debajo de la diagonal principal.

8.5.6. pvsyr2k

`c=PyPBLAS.pvsyr2k(alpha,a,b,beta,c[,uplo,trans])`

La función '`PyPBLAS.pvsyr2k`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices son simétricas:

$$\alpha \cdot A \times B^T + \alpha \cdot B \times A^T + beta \cdot C \rightarrow C$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvsyr2k` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices simétricas con elementos de tipo real o complejo. Un detalle importante en esta rutina es la suposición que las matrices de entrada son simétricas, de este modo se realizan los cálculos con la parte superior (o inferior) a la diagonal. El resultado válido en la matriz devuelta por la función será el situado en la posición que le indiquemos a través del parámetro `uplo`. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times m$.
  - '`b`': Matriz de dimensiones $m \times m$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times m$.

- 'uplo': Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
  - uplo='U': (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
  - uplo='L': Se obtienen los resultados en la diagonal principal y por debajo de ella.
- 'trans': A partir de este parámtetro se pueden realizar dos operaciones diferentes, realizando (o no) la transpuesta de la matriz a y b:
  - trans='N': (Valor por defecto).La operación que se lleva a cabo es : $\alpha \cdot A \times B^T + \alpha \cdot B \times A^T + beta \cdot C \rightarrow C$.
  - trans='T': La operación que se lleva a cabo es : $\alpha \cdot A^T \times B + \alpha \cdot B^T \times A + beta \cdot C \rightarrow C$.

- Parametros de Salida
  - 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
def make_sym_b(x,y):
        return x*y
def make_sym_c(x,y):
        return x+y
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvSYR2K"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([n,n])
        print "a=",a
        b=fromfunction(make_sym_b,(n,n))
        print "b=",b
        c=fromfunction(make_sym_c,(n,n))
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,b,beta,c=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvsyr2k(alpha,a,b,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvSYR2K=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvsyr2k.py
Example of using PyPBLAS 3: PvSYR2K
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
b= [[ 0  0  0  0  0  0]
 [ 0  1  2  3  4  5]
 [ 0  2  4  6  8 10]
 [ 0  3  6  9 12 15]
 [ 0  4  8 12 16 20]
 [ 0  5 10 15 20 25]]
c= [[ 0  1  2  3  4  5]
 [ 1  2  3  4  5  6]
 [ 2  3  4  5  6  7]
 [ 3  4  5  6  7  8]
 [ 4  5  6  7  8  9]
 [ 5  6  7  8  9 10]]
alpha= 2.0 ; beta= 3.0
PvSYR2K= [[   0.    1.    2.    3.    4.    5.]
 [  33.   66.    3.    4.    5.    6.]
 [  66.   99.  132.    5.    6.    7.]
 [  99.  132.  165.  198.    7.    8.]
 [ 132.  165.  198.  231.  264.    9.]
 [ 165.  198.  231.  264.  297.  330.]]
```

Podemos ver en este ejemplo, que el resultado no es una matriz simétrica aunque debiera serlo puesto que `a` y `c` lo son y el resultado de la operación descrita ha de genera una matriz simétrica. Como por defecto `uplo='U'`, el resultado es una matriz simétrica con los valores correctos situados en y por encima de la diagonal. Si ejecutaramos `c=PyPBLAS.pvsyr2k(alpha,a,b,beta,c,uplo='L')`, podríamos comprobar que los valores correctos se obtienen en y por debajo de la diagonal principal.

8.5.7. pvher2k

`c=PyPBLAS.pvher2k(alpha,a,b,beta,c[,uplo,trans])`

La función '`PyPBLAS.pvher2k`' implementa la siguiente operación entre matrices teniendo en cuenta que este tipo de matrices son simétricas y con elementos de tipo complejo:

$$\alpha \cdot A \times B^H + \alpha \cdot B \times A^H + beta \cdot C \rightarrow C$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvher2k` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices simétricas con elementos de tipo complejo. Un detalle importante en esta rutina es la suposición que las matrices de entrada son simétricas, de este modo se realizan los cálculos con la parte superior (o inferior) a la diagonal. El resultado válido en la matriz devuelta por la función será el situado en la posición que le indiquemos a través del parámetro `uplo`. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`

- 'alpha':Escalar.
- 'a': Matriz de dimensiones $m \times m$.
- 'b': Matriz de dimensiones $m \times m$.
- 'beta':Escalar.
- 'c': Matriz de dimensiones $m \times m$.
- 'uplo': Debido a que las matrices son simétricas, con este caracter indicamos en que lado de la diagonal principal vamos a realizar cálculos y obtener los resultados:
  - uplo='U': (Valor por defecto).Se obtienen los resultados en la diagonal principal y por encima de ella.
  - uplo='L': Se obtienen los resultados en la diagonal principal y por debajo de ella.
- 'trans': A partir de este parámtetro se pueden realizar dos operaciones diferentes, realizando (o no) la transpuesta a la matriz a:
  - trans='N': (Valor por defecto).La operación que se lleva a cabo es :$\alpha \cdot A \times B^H + \alpha \cdot B \times A^H + beta \cdot C \rightarrow C$.
  - trans='H': La operación que se lleva a cabo es : $\alpha \cdot A^H \times B + \alpha \cdot B^H \times A + beta \cdot C \rightarrow C$.

- Parametros de Salida
  - 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
#Dimension of Arrays
def make_sym_b(x,y):
        return x*y +(x+y)*1j
def make_sym_c(x,y):
        return x+y +(x*y)*1j
n=6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvHER2K"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=ones([n,n])+1j*ones([n,n])
        print "a=",a
        b=fromfunction(make_sym_b,(n,n))
        print "b=",b
        c=fromfunction(make_sym_c,(n,n))
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,b,beta,c=None,None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvsyr2k(alpha,a,b,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvHER2K=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvher2k.py
Example of using PyPBLAS 3: PvHER2K
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]
 [ 1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j  1.+1.j]]
b= [[  0. +0.j   0. +1.j   0. +2.j   0. +3.j   0. +4.j   0. +5.j]
 [  0. +1.j   1. +2.j   2. +3.j   3. +4.j   4. +5.j   5. +6.j]
 [  0. +2.j   2. +3.j   4. +4.j   6. +5.j   8. +6.j  10. +7.j]
 [  0. +3.j   3. +4.j   6. +5.j   9. +6.j  12. +7.j  15. +8.j]
 [  0. +4.j   4. +5.j   8. +6.j  12. +7.j  16. +8.j  20. +9.j]
 [  0. +5.j   5. +6.j  10. +7.j  15. +8.j  20. +9.j  25.+10.j]]
c= [[  0. +0.j   1. +0.j   2. +0.j   3. +0.j   4. +0.j   5. +0.j]
 [  1. +0.j   2. +1.j   3. +2.j   4. +3.j   5. +4.j   6. +5.j]
 [  2. +0.j   3. +2.j   4. +4.j   5. +6.j   6. +8.j   7.+10.j]
 [  3. +0.j   4. +3.j   5. +6.j   6. +9.j   7.+12.j   8.+15.j]
 [  4. +0.j   5. +4.j   6. +8.j   7.+12.j   8.+16.j   9.+20.j]
 [  5. +0.j   6. +5.j   7.+10.j   8.+15.j   9.+20.j  10.+25.j]]
alpha= 2.0 ; beta= 3.0
PvHER2K= [[  60. +0.j    1. +0.j    2. +0.j    3. +0.j    4. +0.j    5. +0.j]
 [ 105.+18.j  150. +0.j    3. +2.j    4. +3.j    5. +4.j    6. +5.j]
 [ 150.+36.j  195.+24.j  240. +0.j    5. +6.j    6. +8.j    7.+10.j]
 [ 195.+54.j  240.+45.j  285.+36.j  330. +0.j    7.+12.j    8.+15.j]
 [ 240.+72.j  285.+66.j  330.+60.j  375.+54.j  420. +0.j    9.+20.j]
 [ 285.+90.j  330.+87.j  375.+84.j  420.+81.j  465.+78.j  510. +0.j]]
```

Podemos ver en este ejemplo, que el resultado no es una matriz simétrica aunque debiera serlo puesto que `a` y `c` lo son y el resultado de la operación descrita ha de genera una matriz simétrica. Como por defecto `uplo='U'`, el resultado es una matriz simétrica con los valores correctos situados en y por encima de la diagonal. Si ejecutaramos `c=PyPBLAS.pvher2k(alpha,a,b,beta,c,uplo='L')`, podríamos comprobar que los valores correctos se obtienen en y por debajo de la diagonal principal.

8.5.8. pvtran

`c=PyPBLAS.pvtran(alpha,a,beta,c)`

La función '`PyPBLAS.pvtran`' implementa la siguiente operación entre matrices con elementos de tipo real:

$$\beta \cdot C + \alpha \cdot A^T + \rightarrow C$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtran` informará de ello y no podrá finalizar la operación.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $n \times m$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times n$.
- `Parametros de Salida`

- 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
m,n=8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvTRAN"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*n),[n,m])
        print "a=",a
        c=ones([m,n])
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,beta,c=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvtran(alpha,a,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvTRAN=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtran.py
Example of using PyPBLAS 3: PvTRAN
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
c= [[1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
alpha= 2.0 ; beta= 3.0
PvTRAN= [[  3.   5.   7.   9.  11.  13.  15.  17.]
 [ 19.  21.  23.  25.  27.  29.  31.  33.]
 [ 35.  37.  39.  41.  43.  45.  47.  49.]
 [ 51.  53.  55.  57.  59.  61.  63.  65.]
 [ 67.  69.  71.  73.  75.  77.  79.  81.]
 [ 83.  85.  87.  89.  91.  93.  95.  97.]]
```

En el ejemplo mostrado podemos ver como generamos las matrices `a` y `c` con las dimensiones adecuadas para poder realizar la operación, por tanto `a` es $n \times m$ y `b` es $m \times n$ resultado una matriz de tamaño $m \times n$.

8.5.9. pvtranu

`c=PyPBLAS.pvtranu(alpha,a,beta,c)`

La función '`PyPBLAS.pvtranu`' implementa la siguiente operación entre matrices con elementos de tipo complejo:

$$\beta \cdot C + \alpha \cdot A^T + \rightarrow C$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtranu` informará de ello y no podrá finalizar la operación.

Se ha de tener en cuenta que los vectores,matrices y escalares de entrada han de ser de tipo PyACTS. Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $n \times m$.
  - '`beta`':Escalar.
  - '`c`': Matriz de dimensiones $m \times n$.
- `Parametros de Salida`
  - '`c`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
m,n=8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvTRANU"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*n),[n,m])*(1+1.j)
        print "a=",a
        c=ones([m,n])*(1-1.j)
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,beta,c=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvtranu(alpha,a,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvTRANU=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtranu.py
Example of using PyPBLAS 3: PvTRANU
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[  0. +0.j   1. +1.j   2. +2.j   3. +3.j   4. +4.j   5. +5.j
         6. +6.j   7. +7.j]
 [  8. +8.j   9. +9.j  10.+10.j  11.+11.j  12.+12.j  13.+13.j
        14.+14.j  15.+15.j]
 [ 16.+16.j  17.+17.j  18.+18.j  19.+19.j  20.+20.j  21.+21.j
        22.+22.j  23.+23.j]
 [ 24.+24.j  25.+25.j  26.+26.j  27.+27.j  28.+28.j  29.+29.j
        30.+30.j  31.+31.j]
 [ 32.+32.j  33.+33.j  34.+34.j  35.+35.j  36.+36.j  37.+37.j
        38.+38.j  39.+39.j]
 [ 40.+40.j  41.+41.j  42.+42.j  43.+43.j  44.+44.j  45.+45.j
        46.+46.j  47.+47.j]]
c= [[ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]]
alpha= 2.0 ; beta= 3.0
PvTRANU= [[  3. -3.j   5. -1.j   7. +1.j   9. +3.j  11. +5.j  13. +7.j
        15. +9.j  17.+11.j]
 [ 19.+13.j  21.+15.j  23.+17.j  25.+19.j  27.+21.j  29.+23.j
        31.+25.j  33.+27.j]
 [ 35.+29.j  37.+31.j  39.+33.j  41.+35.j  43.+37.j  45.+39.j
        47.+41.j  49.+43.j]
 [ 51.+45.j  53.+47.j  55.+49.j  57.+51.j  59.+53.j  61.+55.j
        63.+57.j  65.+59.j]
 [ 67.+61.j  69.+63.j  71.+65.j  73.+67.j  75.+69.j  77.+71.j
        79.+73.j  81.+75.j]
 [ 83.+77.j  85.+79.j  87.+81.j  89.+83.j  91.+85.j  93.+87.j
        95.+89.j  97.+91.j]]
```

En el ejemplo mostrado podemos ver como generamos las matrices `a` y `c` con las dimensiones adecuadas para poder realizar la operación , por tanto `a` es $n \times m$ y `b` es $m \times n$ resultado una matriz de tamaño $m \times n$.

8.5.10. pvtranc

`c=PyPBLAS.pvtranc(alpha,a,beta,c)`

La función '`PyPBLAS.pvtranc`' implementa la siguiente operación entre matrices con elementos de tipo complejo:

$$\beta \cdot C + \alpha \cdot A^H + \rightarrow C$$

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtranc` informará de ello y no podrá finalizar la operación.

Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.

- 'a': Matriz de dimensiones $n \times m$.
- 'beta':Escalar.
- 'c': Matriz de dimensiones $m \times n$.

- Parametros de Salida
  - 'c': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
m,n=8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvTRANC"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*n),[n,m])*(1+1.j)
        print "a=",a
        c=ones([m,n])*(1-1.j)
        print "c=",c
        alpha,beta=2.,3.
        print "alpha=",alpha,";","beta=",beta
else:
        alpha,a,beta,c=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
beta=Scal2PyACTS(beta,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call PBLAS routine
c= PyPBLAS.pvtranc(alpha,a,beta,c)
c_num=PyACTS2Num(c)
if PyACTS.iread==1:
        print "PvTRANC=",transpose(c_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtranc.py
Example of using PyPBLAS 3: PvTRANC
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[  0. +0.j   1. +1.j   2. +2.j   3. +3.j   4. +4.j   5. +5.j
         6. +6.j   7. +7.j]
 [  8. +8.j   9. +9.j  10.+10.j  11.+11.j  12.+12.j  13.+13.j
        14.+14.j  15.+15.j]
 [ 16.+16.j  17.+17.j  18.+18.j  19.+19.j  20.+20.j  21.+21.j
        22.+22.j  23.+23.j]
 [ 24.+24.j  25.+25.j  26.+26.j  27.+27.j  28.+28.j  29.+29.j
        30.+30.j  31.+31.j]
 [ 32.+32.j  33.+33.j  34.+34.j  35.+35.j  36.+36.j  37.+37.j
        38.+38.j  39.+39.j]
 [ 40.+40.j  41.+41.j  42.+42.j  43.+43.j  44.+44.j  45.+45.j
        46.+46.j  47.+47.j]]
c= [[ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]
 [ 1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j  1.-1.j]]
alpha= 2.0 ; beta= 3.0
PvTRANU= [[  3. -3.j   5. -5.j   7. -7.j   9. -9.j  11.-11.j  13.-13.j
        15.-15.j  17.-17.j]
 [ 19.-19.j  21.-21.j  23.-23.j  25.-25.j  27.-27.j  29.-29.j
        31.-31.j  33.-33.j]
 [ 35.-35.j  37.-37.j  39.-39.j  41.-41.j  43.-43.j  45.-45.j
        47.-47.j  49.-49.j]
 [ 51.-51.j  53.-53.j  55.-55.j  57.-57.j  59.-59.j  61.-61.j
        63.-63.j  65.-65.j]
 [ 67.-67.j  69.-69.j  71.-71.j  73.-73.j  75.-75.j  77.-77.j
        79.-79.j  81.-81.j]
 [ 83.-83.j  85.-85.j  87.-87.j  89.-89.j  91.-91.j  93.-93.j
        95.-95.j  97.-97.j]]
```

En el ejemplo mostrado podemos ver como generamos las matrices `a` y `c` con las dimensiones adecuadas para poder realizar la operación, por tanto `a` es $n \times m$ y `b` es $m \times n$ resultado una matriz de tamaño $m \times n$.

8.5.11. pvtrmm

`b=PyPBLAS.pvtrmm(alpha,a,b[,side='L',uplo='U',transa='N',diag='N'])`

La función '`PyPBLAS.pvtrmm`' implementa la siguiente operación entre matrices asumiendo que las matrices `a` y `b` son matrices triangulares:

$$\alpha \cdot op(A) \times B \rightarrow B; \alpha \cdot B \times op(A) \rightarrow B$$

Donde $op(A)$ depende del valor del parámetro '`transa`':

- `trans='N'`: No se se realiza ninguna operación sobre la matriz.
- `trans='T'`: Se realiza la transpuesta de A: A^T.

- `trans='C'` : Se realiza la transpuesta conjugada de A: A^H.

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtrmm` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices con elementos de tipo real y complejo.Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - '`alpha`':Escalar.
  - '`a`': Matriz de dimensiones $m \times n$.
  - '`b`': Matriz de dimensiones $m \times n$.
  - '`side`': Caracter que indica la posición de la matriz `a` en la operación:
    - `side='L'` : (Valor por defecto).La operación que se realiza es: $\alpha \cdot op(A) \times B \rightarrow B$.
    - `side='R'` : La operación que se realiza es $\alpha \cdot B \times op(A) \rightarrow B$.
  - '`uplo`': Debido a que las matrices son triangulares, con este caracter indicamos en que lado de la diagonal principal se encuentran los datos validos , es decir, si la matriz es triangular superior o inferior :
    - `uplo='U'` : (Valor por defecto).Las matrices son triangulares superiores
    - `uplo='L'` : Las matrices son triangulares inferiores
  - '`transa`': Caracter que indica la opearación a realizar sobre la matriz `a`:
    - `transa='N'`:No transpuesta, es decir, no se realiza operación.(valor por defecto)
    - `transa='T'`:Transpuesta.
    - `transa='C'` : Transpuesta conjugada.
  - '`diag`': Caracter que indica si la matriz triangular es unitaria o no:
    - `diag='N'` : Matriz triangular NO unitaria (valor por defecto).
    - `diag='U'` : Matriz triangular Unitaria.
  - `Parametros de Salida`
    - '`b`': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
m,n=8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvTRMM"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=reshape(range(m*m),[m,m])
        print "a=",a
        b=reshape(range(m*n),[m,n])
        print "b=",b
        alpha=2.
        print "alpha=",alpha
else:
        alpha,a,b=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
b= PyPBLAS.pvtrmm(alpha,a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
        print "PvTRMM=",transpose(b_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtrmm.py
Example of using PyPBLAS 3: PvTRMM
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]
 [48 49 50 51 52 53 54 55]
 [56 57 58 59 60 61 62 63]]
b= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]
 [36 37 38 39 40 41]
 [42 43 44 45 46 47]]
alpha= 2.0
PvTRMM= [[  1680.   4368.   6852.   8820.   9960.   9960.   8508.   5292.]
 [  1736.   4536.   7098.   9110.  10260.  10236.   8726.   5418.]
 [  1792.   4704.   7344.   9400.  10560.  10512.   8944.   5544.]
 [  1848.   4872.   7590.   9690.  10860.  10788.   9162.   5670.]
 [  1904.   5040.   7836.   9980.  11160.  11064.   9380.   5796.]
 [  1960.   5208.   8082.  10270.  11460.  11340.   9598.   5922.]]
```

Los paramentros `side`, `uplo` ,`transa` y `diag` son parámetros opcionales que tienen un valor establecido por defecto. En el caso que no se especifiquen estos parámetros tomaraán sus valores por defecto. Para especificar un valor diferente se puede realizar del siguiente modo:

```
b=PyPBLAS.pvtrmm(alpha,a,b,side='R',uplo='L')
```

8.5.12. pvtrsm

`b=PyPBLAS.pvtrsm(alpha,a,b[,side='L',uplo='U',transa='N',diag='N'])`

La función '`PyPBLAS.pvtrsm`' implementa la siguiente operación entre matrices asumiendo que las matrices `a` y `b` son matrices triangulares:

$$\alpha \cdot op(A^-1) \times B \rightarrow B; \alpha B \cdot B \times op(A^-1) \rightarrow B$$

Donde $op(A)$ depende del valor del parámetro '`transa`':

- `trans='N'`: No se se realiza ninguna operación sobre la matriz.
- `trans='T'`: Se realiza la transpuesta de A: A^T.
- `trans='C'`: Se realiza la transpuesta conjugada de A: A^H.

En el caso que las dimensiones de alguna de las entradas no sea correcta, la rutina `PyPBLASpvtrmm` informará de ello y no podrá finalizar la operación.

Esta rutina se provee para matrices con elementos de tipo real y complejo.Las características de cada uno de los parámetros son las siguientes:

- `Parametros de Entrada`
  - 'alpha':Escalar.
  - 'a': Matriz de dimensiones $m \times n$.
  - 'b': Matriz de dimensiones $m \times n$.
  - 'side': Caracter que indica la posición de la matriz a en la operación:
    - side='L': (Valor por defecto).La operación que se realiza es: $\alpha \cdot op(A^-1) \times B \rightarrow B$.
    - side='R': La operación que se realiza es $\alpha \cdot B \times op(A^-1) \rightarrow B$.
  - 'uplo': Debido a que las matrices son triangulares, con este caracter indicamos en que lado de la diagonal principal se encuentran los datos validos , es decir, si la matriz es triangular superior o inferior :
    - uplo='U': (Valor por defecto).Las matrices son triangulares superiores
    - uplo='L': Las matrices son triangulares inferiores
  - 'transa': Caracter que indica la opearación a realizar sobre la matriz a:
    - transa='N':No transpuesta, es decir, no se realiza operación.(valor por defecto)
    - transa='T':Transpuesta.
    - transa='C': Transpuesta conjugada.
  - 'diag': Caracter que indica si la matriz triangular es unitaria o no:
    - diag='N': Matriz triangular NO unitaria (valor por defecto).
    - diag='U': Matriz triangular Unitaria.
  - `Parametros de Salida`
    - 'b': Vector distribuido donde se almacena el resultado.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
from RandomArray import *
from Numeric import *
m,n=8,6
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using PyPBLAS 3: PvTRSM"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Block's size:",PyACTS.mb,"*",PyACTS.nb
        a=2*identity(m,Float)
        print "a=",a
        b=reshape(range(m*n),[m,n])
        print "b=",b
        alpha=2.
        print "alpha=",alpha
else:
        alpha,a,b=None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
alpha=Scal2PyACTS(alpha,ACTS_lib)
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
b= PyPBLAS.pvtrsm(alpha,a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
        print "PvTRSM=",transpose(b_num)
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPypvtrsm.py
Example of using PyPBLAS 3: PvTRSM
N= 6 ;nprow x npcol: 2 x 2
Block's size: 2 * 2
a= [[ 2.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  2.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  2.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  2.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  2.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  2.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  2.]]
b= [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]
 [36 37 38 39 40 41]
 [42 43 44 45 46 47]]
alpha= 2.0
PvTRSM= [[  0.   6.  12.  18.  24.  30.  36.  42.]
 [  1.   7.  13.  19.  25.  31.  37.  43.]
 [  2.   8.  14.  20.  26.  32.  38.  44.]
 [  3.   9.  15.  21.  27.  33.  39.  45.]
 [  4.  10.  16.  22.  28.  34.  40.  46.]
 [  5.  11.  17.  23.  29.  35.  41.  47.]]
```

Los paramentros `side`, `uplo` ,`transa` y `diag` son parámetros opcionales que tienen un valor establecido por defecto. En el caso que no se especifiquen estos parámetros tomaraán sus valores por defecto. Para especificar un valor diferente se puede realizar del siguiente modo:

```
b=PyPBLAS.pvtrsm(alpha,a,b,side='R',uplo='L')
```

CAPÍTULO

NINE

PyScaLAPACK

En el presente capítulo realizaremos una breve introducción a la problemática que intentan resolver el conjunto de rutinas incluidas en la librería ScaLAPACK. Posteriormente, describiremos cada una de las rutinas incluidas en PyScaLAPACK y describiremos una a una su funcionamiento, Parámetros de entrada y salida y mostraremos un ejemplo con los resultados mostrados en pantalla.

9.1. Rutinas sencillas para Ecuaciones Lineales

9.1.1. pvgesv

`b,info=pvgesv(a,b[,ia=1,ja=1,ib=1,jb=1])`

La rutina '`pvgesv`' resuelve la ecuacion del sistema lineal siguiente:

$$A \times X = B$$

La resolución de este sistema de ecuaciones se lleva a cabo mediante la descomposición LU utilizando la pivotacion parcial y el intercambio de filas de A. De este modo obtenemos `sub( A ) = P * L * U`, donde `P` es una matriz permutada, `L` es una matriz triangular unitaria, y `U` es una matriz triangular supperior. La forma factorizada de A se utiliza para resolver entonces este múltiple sistema de ecuaciones `sub( A ) * X = sub( B )`. El vector de salida es un vector donde cada columna representa la solución a cada uno de los sistemas planteados mediante la matriz `b` de entrada.

La forma de resolver este sistema lineal depende de las características de la matriz y deberemos usar una rutina u otra de esta sección dependiendo del tipo de matriz. En este manual, trataremos de describir de una forma sencilla y clara las principales características de cada uno de los métodos de resolución de este sistema lineal.

Para matrices de tipo general, la resolución se realiza mediante factorización LU con pivotación parcial:

$$A = PLU$$

donde P es una matriz permutación, L es una matriz triangular inferior con los elementos de la diagonal unitarios, y U es una matriz triangular superior (trapezoidal superior si m<n)

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes dimensiones $n \times n$.
  - `b`: Vector de dimensiones $n \times nrhs$.

- Parámetros de Salida
  - b: Vector distribuido donde se almacena el resultado.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
    - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Example of using ScaLAPACK: PvGESV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
        b=ones((n,nrhs))
        print "b'=",transpose(b)
else:
        a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
b,info= PySLK.pvgesv(a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
        print "a*x=b --> x'=",transpose(b_num)
        print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvgesv.py
Example of using ScaLAPACK: PvGESV
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [[1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [[ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]
 [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
Info: 0
```

9.1.2. pvdbsv

`b,info=PyScaLAPACK.pvdbsv(a,b[bwl=n-1,bwu=n-1,ia=1,ja=1,ib=1,jb=1])`

La rutina '`PyScaLAPACK.pvdbsv`' resuelve una ecuación del siguiente tipo:

$$A \times X = B$$

donde A es una matriz $N \times N$ real o compleja distribuida por bandas dominantes en su diagonal indicadas por los parámetros `bwl` y `bwu`. La eliminación Gaussiana sin pivotación se utiliza para reordenar la matriz a partir de su factorización `L U`.

Esta rutina se provee para matrices con elementos de tipo real y complejo.Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `b`: Vector de dimensiones $n \times nrhs$.
  - `bwl`: Número de subdiagonales inferiores $0 <= bwl <= n-1$. Por defecto, $bwl = n-1$.
  - `bwu`: Número de subdiagonales superiores $0 <= bwu <= n-1$.Por defecto, $bwu = n-1$.
- `Parámetros de Salida`
  - `b`: Vector distribuido donde se almacena el resultado.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

El resultado de este código es el siguiente:

9.1.3. pvgbsv

`b,info=PyScaLAPACK.pvgbsv(a,b[bwl=n-1,bwu=n-1,ia=1,ja=1,ib=1,jb=1])`

La rutina '`PyScaLAPACK_pvgbsv`' resuelve una ecuación del siguiente tipo:

$$A * X = B$$

donde A es una matriz $N \times N$ real o compleja distribuida por bandas dominantes en su diagonal indicadas por los parámetros `bwl` y `bwu`. La eliminación Gaussiana con pivotación se utiliza para reordenar la matriz a partir de su factorización `P L U`.

Esta rutina se provee para matrices con elementos de tipo real y complejo.Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `b`: Vector de dimensiones $n \times 1$.
  - `bwl`: Número de subdiagonales inferiores $0 <= bwl <= n-1$.Por defecto, $bwl = n-1$, donde n es el tamaño de la matriz a.
  - `bwu`: Número de subdiagonales superiores $0 <= bwu <= n-1$.Por defecto, $bwu = n-1$, donde n es el tamaño de la matriz a.
- `Parámetros de Salida`
  - `b`: Vector distribuido donde se almacena el resultado.
  - `info`: Resultado global de la ejecución
    - `info= 0`: Ejecución con éxito
    - `info<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entonces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `info<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

El resultado de este código es el siguiente:

9.1.4. pvdtsv

La rutina `pvdtsv` resuelve una ecuación del siguiente tipo:

$$A * X = B$$

donde A es una matriz $N \times N$ real o compleja tridiagonal, es decir, con valores únicamente en la diagonal principal y en la diagonal superior e inferior contigua a la principal.

Esta rutina se provee para matrices con elementos de tipo real y complejo.Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `b`: Vector de dimensiones $n \times 1$.
  - `bwl`: Número de subdiagonales inferiores $0 <= bwl <= n - 1$.Por defecto, $bwl = n - 1$, donde n es el tamaño de la matriz a.
  - `bwu`: Número de subdiagonales superiores $0 <= bwu <= n - 1$.Por defecto, $bwu = n - 1$, donde n es el tamaño de la matriz a.
- `Parámetros de Salida`
  - `b`: Vector distribuido donde se almacena el resultado.
  - `info`: Resultado global de la ejecución
    - `info= 0`: Ejecución con éxito
    - `info<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entonces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `info<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

El resultado de este código es el siguiente:

9.2. Rutinas sencillas para problemas generales de raices lineales

9.2.1. pvgels

`b,info= PySLK.pvgels(a,b[,trans])`

La rutina `pvgels` resuelve sistemas lineales determinados o indeterminados descritos en una matriz MxN o su traspuesta, usando una factorización QR o LQ de a. Se asume que a tiene rango máximo.

Esta rutina se provee para matrices con elementos de tipo real y complejo.

Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times m$.
  - `b`: Matriz de dimensiones $m \times nrhs$.
  - `trans`: Operación que se realiza sobre la matriz a.
    - Si `trans='N'` y `m=n` : Encuentra la raiz al sistema determinado, por ejemplo resuelve el problema `minimize` $||B - A * X||$.
    - Si `trans='N'` y `m<n` : Encuentra la solución mínima del sistema indeterminado $A * X = B$
    - Si `trans='T'` y `m>=n` : Encuentra la solución mínima del sistema indeterminado $A^T * X = B$
    - Si `trans='T'` y `m<n` : Encuentra la raiz al sistema determinado, por ejemplo resuelve el problema `minimize` $||B - A^T * X||$.
- `Parámetros de Salida`
  - `b`: Vector de tipo PyACTS donde se almacena el resultado.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es un vector y la entrada j-esima tuvo un valor ilegal, entonces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal, entonces `info=-1`.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvGELS"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
        b=ones((n,nrhs))
        print "b'=",transpose(b)
else:
        a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
b,info= PySLK.pvgels(a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
        print "a*x=b --> x'=",transpose(b_num)
        print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvgels.py
Ejemplo de Utilizacion ScaLAPACK: PvGELS
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [[1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [[ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]
 [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
Info: 0
```

9.3. Rutinas sencillas para problemas de valores únicos

9.3.1. pvsyev

`w,z,info= PySLK.pvsyev(a,jobz,uplo)`

La rutina `pvsyev` resuelve todos los valores únicos y, opcionalmente, los vectores únicos de una matriz A **simétrica** de tipo real llamando a una secuencia adecuada de las rutinas de PyScaLAPACK.

`pvsyev` asume un sistema homogeneo y no realiza las comprobaciones de la consistencia de los valores unicos y de los vectores unicos en los diferentes procesos. Por tanto, es posible que un sistema heterogéneo pueda devolver resultados incorrectos sin mensajes de error.

Esta rutina se provee para matrices con elementos únicamente de tipo real.

Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz simétrica de dimensiones $n \times n$.
  - `jobz`: Indica si la rutina ha de calcular o no los vectores únicos.:
    - `jobz='N'`: Calcula únicamente los valores únicos.
    - `jobz='V'`: Calcula los valores únicos y los vectores únicos.
  - `uplo`: Especifica la parte de la matriz a que es utilizada para realizar el cálculo:
    - `uplo='U'`: Únicamente la parte superior triangular de la matriz a es utilizada para definir los elementos de la matriz.
    - `uplo='L'`: Únicamente la parte inferior triangular de la matriz a es utilizada para definir los elementos de la matriz.
- `Parámetros de Salida`
  - `w`: Vector de tipo **Numeric** de tamaño n donde se devuelven los valores singulares de la matriz a.
  - `z`: Matriz de tipo **PyACTS** de tamaño $n \times n$ donde se devuelven los vectores singulares de la matriz a.
  - `info`: Resultado global de la ejecución

- = 0: Ejecución con éxito
- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entonces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal, entonces info=-i.
- >0: Si info tiene un valor de 1 a n, entonces el i-esimo valor único no converge despues de un total de 30*N iteraciones. Si info=n+1, entonces se ha detectado heterogeneidad en los valores únicos.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvSYEV"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
else:
        a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
w,z,info= PySLK.pvsyev(a,jobz='V')
z_num=PyACTS2Num(z)
if PyACTS.iread==1:
        print "Eigenvalues --> w'=",transpose(w)
        print "Eigenvectors --> z=",z_num
        print "Info:",info
PyACTS.gridexit()
```

De esta rutina queremos destacar el hecho que w es una matriz de tipo Numeric y será conocida por todos los procesos que intervienen en la malla. Sin embargo, z es una matriz de tipo PyACTS, lo que implica que sus valores se encuentran distribuidos entre los distintos procesos que intervienen en la malla. En el caso que quisieramos obtener toda los valores en un único proceso, éste debería llamar a la función PyACTS2Num tal y como se indica en este ejemplo. El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvsyev.py
Ejemplo de Utilizacion ScaLAPACK: PvSYEV
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Eigenvalues --> w'= [ 8.  8.  8.  8.  8.  8.  8.  8.]
Eigenvectors --> z= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
Info: 0
```

9.3.2. pvgesvd

`s,u,vt,info= pvgesvd(a[,jobu,jobvt])`

La rutina `pvgesvd` calcula la descomposición en valores singulares (SVD) de una matriz A de tamaño $M \times N$, opcionalmente calcula los vectores singulares por la izquierda y/o por la derecha. La descomposición en valores singulares puede ser expresada como:

$$A = U \times SIGMA \times V^T$$

donde $SIGMA$ es una matriz de tamaño $M \times N$ con todos sus elementos igual a cero menos los `min(M,N)` de su diagonal principal. Los elementos de la diagonal principal de $SIGMA$ son valores únicos de A y las columnas de U y V son los correspondientes vectores únicos por la derecha y por la izquierda respectivamente. Los valores singulares son devueltos en el vector S en orden decreciente y sólo se calculan las primeras `min(M,N)` columnas de U y las primeras `min(M,N)` filas de V^T.

Esta rutina se provee para matrices con elementos únicamente de tipo real.

Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de tipo PyACTS de dimensiones $m \times n$.
  - `jobu`: Indica si la rutina ha de calcular o no la matriz U:
    - `jobu='N'`: No calcula los vectores singulares por la izquierda.
    - `jobu='V'`: Calcula las primeras `min(m,n)` columnas de U. Es decir los vectores singulares por la izquierda.
  - `jobvt`: Indica si la rutina ha de calcular o no la matriz U:
    - `jobvt='N'`: No calcula la matriz V^T, que se corresponde con los vectores singulares por la derecha.

- jobvt='V': Calcula las primeras min(m,n) filas de V^T. Es decir los vectores singulares por la derecha.

- Parámetros de Salida

  - s: Vector de tipo **Numeric** de tamaño min(m,n) donde se devuelven los valores singulares de la matriz a.
  - u: Matriz de tipo **PyACTS** donde se devuelven los vectores singulares de la matriz por la izquierda.
  - vt: Matriz de tipo **PyACTS** donde se devuelven los vectores singulares de la matriz por la derecha.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo tuvo un valor ilegal, entonces info=-i.
    - >0: No converge el cálculo. Si info=n+1, entonces se ha detectado heterogeneidad en los valores únicos y la exactitud de los resultados no puede ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvGESVD"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
else:
        a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
s,u,vt,info= PySLK.pvgesvd(a,jobu='V',jobvt='V')
u_num=PyACTS2Num(u)
vt_num=PyACTS2Num(vt)
if PyACTS.iread==1:
        print "Singular Values--> s'=",transpose(s)
        print "Vectores Ortogonales --> u=",u_num
        print "Vectores Ortogonales --> vt=",vt_num
        print "Info:",info
PyACTS.gridexit()
```

De esta rutina queremos destacar el hecho que s es una matriz de tipo Numeric y será conocida por todos los procesos que intervienen en la malla. Sin embargo, u,y vt son matrices de tipo PyACTS, lo que implica que sus valores se encuentran distribuidos entre los distintos procesos que intervienen en la malla. En el caso que quisieramos obtener toda los valores en un único proceso, éste debería llamar a la función PyACTS2Num tal y como se indica en este ejemplo. El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvgesvd.py
Ejemplo de Utilizacion ScaLAPACK: PvGESVD
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Singular Values--> s'= [ 8.  8.  8.  8.  8.  8.  8.  8.]
Vectores Ortogonales --> u= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
Vectores Ortogonales --> vt= [[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
Info: 0
```

9.4. Rutinas especializadas para ecuaciones lineales

9.4.1. pvgesvx

`a,af,equed,r,c,b,x,rcond,ferr,berr,info= PySLK.pvgesvx(a,b[,fact,trans,equed,ACTS_af,ACT`

La rutina '`pvgesvx`' resuelve la ecuacion del sistema lineal siguiente:

$$A \times X = B$$

donde A es una matriz $n \times n$, y B y X son matrices de tamaño $n \times nrhs$. Los errores en la solución y las condiciones de la estimación tambien se especifican en esta rutina.

La utilización de estas rútinas es mas complejas que las rutinas sencillas (9.1), vistas en apartados anteriores. En esta rutina se realizan los siguientes pasos:

1. Si `fact='E'`, con el fin de equilibrar el sistema se computan los factores reales escalares.
   - Si `trans='N'`: $diag(R) \times A \times diag(C) \times inv(diag(C)) \times X = diag(R) \times B$
   - Si `trans='T'`: $(diag(R) \times A \times diag(C))^T \times inv(diag(R)) \times X = diag(C) \times B$
   - Si `trans='C'`: $(diag(R) \times A \times diag(C))^H \times inv(diag(R)) \times X = diag(C) \times B$

   Que el sistema sea o no equilibrado depende de la matriz A, pero si la equilibración es utilizada , A se sobreescribirá por $diag(R) \times A \times diag(C)$ y B por $diag(R) \times B$ (si `trans='N'`) o por $diag(C) \times B$ (si `trans='T'`).
2. `fact='E'`, o `fact='N'`, se utiliza la descomposición LU para factorizar la matriz A como:
   $A = P \times L \times U$
   , donde P es una matriz permutación, L es una matriz unitaria triangular inferior, y U es una matriz triangular.
3. La forma factorizada de A se utiliza para estimar el número de condición de la matriz A. Si el número de condición es menos que la precisión de la maquina, los tres siguientes pasos se saltan.
4. El sistema de ecuaciones se resuelve utilizando la forma factorizada de A.
5. Se aplica un proceso iteratico para obtener exactitud en la solución y calcular los errores.
6. Si `fact='E'` y se utiliza equilibración, la matriz X se multiplica previamente por $diag(C)$ (si `trans='N'`) o por $diag(R)$ (si `trans='T' o 'C'`). De este modo se resuelve la ecuación antes de la equilibración.

pvgesvx(ACTS_a,ACTS_b,fact='N',trans='N',equed='N',x=None,ACTS_af=None,rcond=0

- `Parámetros de Entrada`
  - `a`: Matriz de tipo PyACTS de dimensiones $n \times n$.
  - `b`: Matriz de tipo PyACTS de dimensiones $n \times nrhs$.
  - `fact`: Indicaremos si la entra a está en su forma factorizada o no, y en el caso que no lo esté si debemos equilibrarla antes de realizar la factorización.
    - `fact='F'`: La matriz `af` contendrá la forma factorizada de `a`. Si `equed<>'N'`, la matriz `a` ha sido equilibrada con los factores dados por `r` y `c`.
    - `fact='N'`: La matriz `a` se copiará en la matriz `af` y se factorizará.
    - `fact='E'`: La matriz `a` será equilibrada si es necesario y entonces se copiará y factorizará en `af`.
  - `trans`: Indica la forma del sistema de ecuaciones:

- trans='N': $A \times X = B$ (No traspuesta).
- trans='T': $A^T \times X = B$ (traspuesta).
- trans='C': $A^H \times X = B$ (traspuesta).

- af: Este parametro es opcional. Dependiendo de los parámetros de entrada la matriz af tendrá diferente forma.
  - Si fact='F', entonces af es un argumento de entrada y contiene los factores L y U de la factorizacion $A = P \times L \times U$ computada con psgetrf. Si equed='N', entonces af contiene la forma factorizada forma de la matriz a.
  - Si fact='N', entonces af es un argumento de salida que devuelve la forma factorizada L y U de la factorización $A = P \times L \times U$.
  - Si fact='E', entonces af es un argumento de salida que devuelve la forma factorizada L y U de la factorización $A = P \times L \times U$ de la matriz equilibrada a.

x

■ Parámetros de Salida

- a: Como argumento de salida, si equed<>'N' , entonces a es escalada del siguiente modo:
  - Si equed='R' : $A = diag(R) \times A$
  - Si equed='C' : $A = A \times diag(C)$
  - Si equed='B' : $A = diag(R) \times A \times diag(C)$
- af: Como argumento de salida puede tener las siguientes formas:
  - Si fact='N' : entonces af es devuelve los factores L y U de la factorización $A = P \times L \times U$ de la matriz original a.
  - Si fact='E' : entonces af es devuelve los factores L y U de la factorización $A = P \times L \times U$ de la matriz equilibrada a.
- equed: Indica la forma en la que fue realizada la equilibracion:
  - Si equed='N' : No se realizó equilibración.
  - Si equed='R' : Equilibración por filas.
  - Si equed='C' : Equilibración por columnas.
  - Si equed='B' : Ambas equilibraciones fueron realizadas (filas y columnas).
- r: Contiene los factores de escala para las filas.
- c: Contiene los factores de escala para las columnas.
- b: Si equed = 'N', b no se modificará; si trans='N' y equed = 'R' o 'B', b se sobrrescribe por $diag(R) \times B$.
- x: Si info=0, la matriz solución X de tamaño $N \times NRHS$ del sistema de ecuaciones original.
- rcond: Es un número real que expresa la estimación de la equilibración. Si rcond es menor que la precisión de la maquina (rcond=0), la matriz es singular a la precisión de trabajo.
- ferr: La estimación del error "hacia delante" para cada elemento del vector x
- berr: La estimación del error "hacia atrás" para cada elemento del vector x
- info: Indica el estado de salida de la rutina:
  - Si info=0 : Ejecución con éxito
  - Si info<0 : Si info=-i el i-ésimo argumento tuvo un valor ilegal.
  - Si info>0 : Si info=i, y i es
    - <=N : U es exactamente cero. La factorización ha sido completada, pero el factor U es singular.
    - =N+1 : rcond es menor que la precisión de la maquina

Consideramos que son muchos los parametros y las opciones de ejecución en estas rutinas para expertos. A continuación mostramos un ejemplo que resuelve el mismo problema que la rutina pvgesv vista en el apartado anterior.

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvGESVX"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
        b=ones((n,nrhs))
        print "b'=",transpose(b)
else:
        a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
a,af,equed,r,c,b,x,rcond,ferr,berr,info= PySLK.pvgesvx(a,b,fact='E')

x_num=PyACTS2Num(x)
if PyACTS.iread==1:
        print "a*x=b --> x'=",transpose(x_num)
        print "r=",transpose(r)
        print "c=",transpose(c)
        print "ferr:",transpose(ferr),"; berr=",transpose(berr)
        print "Info:",info
PyACTS.gridexit()
```

El resultado en la ejecución de este ejemplo sería el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvgesvx.py
Ejemplo de Utilizacion ScaLAPACK: PvGESVX
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 32 * 32
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [[1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [[ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]
 [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
r= [ [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
c= [ [ 1.  1.  1.  1.  1.  1.  1.  1.]]
ferr: [ [  1.99840144e-15   1.99840144e-15   0.00000000e+00   0.00000000e+00
          0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]] ; berr= [ [ 0.
Info: 0
```

9.4.2. pvposvx

`a,af,equed,sr,sc,b,x,rcond,ferr,berr,info= PySLK.pvposvx(a,b[,fact,uplo,equed,ACTS_af])`

La rutina '`pvposvx`' resuelve la ecuacion del sistema lineal siguiente:

$$A \times X = B$$

donde A es una matriz $n \times n$, y B y X son matrices de tamaño $n \times nrhs$.

Los errores en la solución y las condiciones de la estimación tambien se especifican en esta rutina. La rutina '`pvposvx`' utiliza la factorización de Cholesky

$$A = U \times T \times U$$

o

$$A = L \times L \times T$$

para computar la solución al sistema de ecuaciones anteriormente citado. La utilización de estas rútinas es mas complejas que las rutinas sencillas (9.1), vistas en apartados anteriores. En esta rutina se realizan los siguientes pasos:

1. Si `fact='E'`, con el fin de equilibrar el sistema se computan los factores reales escalares.$diag(SR) \times A \times diag(SC) \times inv(diag(SC)) \times X = diag(SR) \times B$.
2. `fact='E'`, o `fact='N'`, se utiliza la descomposición Cholesky para factorizar la matriz A como:

   $$A = U \times T \times U$$

   o

   $$A = L \times L \times T$$

   donde U es una matriz triangular superior y L es una matriz triangular inferior.

3. La forma factorizada de A se utiliza para estimar el número de condición de la matriz A. Si el número de condición es menos que la precisión de la maquina, los tres siguientes pasos se saltan.
4. El sistema de ecuaciones se resuelve utilizando la forma factorizada de A.
5. Se aplica un proceso iterativo para obtener exactitud en la solución y calcular los errores.
6. Si `fact='E'` y se utiliza equilibración, la matriz X se multiplica previamente por $diag(SR)$, de este modo se resuelve la ecuación antes de la equilibración.

A continuación detallaremos los parámetros de entrada y salida de esta función:

- `Parámetros de Entrada`
  - `a`: Matriz de tipo PyACTS de dimensiones $n \times n$.
  - `b`: Matriz de tipo PyACTS de dimensiones $n \times nrhs$.
  - `fact`: Indicaremos si la matriz `a` está en su forma factorizada o no, y en el caso que no lo esté si debemos equilibrarla antes de realizar la factorización.
    - `fact='F'`: La matriz `af` contendrá la forma factorizada de `a`. Si `equed<>'N'`, la matriz `a` ha sido equilibrada con los factores dados por `r` y `c`.
    - `fact='N'`: La matriz `a` se copiará en la matriz `af` y se factorizará.
    - `fact='E'`: La matriz `a` será equilibrada si es necesario y entonces se copiará y factorizará en `af`.
  - `uplo`: Indica la utilización de la matriz triangular:
    - `uplo='U'`: La matriz triangular superior es almacenada.
    - `uplo='L'`: La matriz triangular inferior es almacenada.
  - `af`: Este parametro es opcional. Dependiendo de los parámetros de entrada la matriz `af` tendrá diferente forma.
    - Si `fact='F'`, entonces `af` es un argumento de entrada y contiene la factorizacion triangular U o L de la factorizacion de Cholesky $A = U \times T \times U$ o $A = L \times L \times T$.
    - Si `fact='N'`, entonces `af` es un argumento de salida que devuelve la forma factorizada L y U de la factorización $A = P \times L \times U$.
    - Si `fact='E'`, entonces `af` es un argumento de salida que devuelve el factor triangular U o L de la factorización $A = U \times T \times U$ o $A = L \times L \times T$ de la matriz equilibrada A.
- `Parámetros de Salida`
  - `a`: Como argumento de salida, si `fact='E'` y `equed='Y'`, entonces `a` se sobreescribe por $A = diag(SR) \times A \times diag(SC)$
  - `af`: Como argumento de salida puede tener las siguientes formas:
    - Si `fact='N'` : entonces `af` devuelve la factorización triangular U o L de Cholesky a partir de $A = U \times T \times U$ o $A = L \times L \times T$ donde A es la matriz original.
    - Si `fact='N'` : entonces `af` devuelve la factorización triangular U o L de Cholesky a partir de $A = U \times T \times U$ o $A = L \times L \times T$ donde A es la matriz equilibrada.
  - `equed`: Indica la forma en la que fue realizada la equilibracion:
    - Si `equed='N'` : No se realizó equilibración.
    - Si `equed='R'` : Equilibración por filas.
    - Si `equed='C'` : Equilibración por columnas.
    - Si `equed='B'` : Ambas equilibraciones fueron realizadas (filas y columnas).
  - `sr`: Contiene los factores de escala para las filas.
  - `sc`: Contiene los factores de escala para las columnas.

- b: Si `equed = 'N'`, b no se modificará; si `trans='N'` y `equed = 'R' o 'B'`, b se sobrrescribe por $diag(R) \times B$. Si `trans='T' o 'C'` y `equed = 'R' o 'B'`, b se sobreescribe por $diage(C) \times B$
- x: Si `info=0`, la matriz solución X de tamaño $N \times NRHS$ del sistema de ecuaciones original.
- rcond: Es un número real que expresa la estimación de la equilibración. Si rcond es menor que la precisión de la maquina (`rcond=0`), la matriz es singular a la precisión de trabajo.
- ferr: La estimación del error "hacia delante" para cada elemento del vector x
- berr: La estimación del error "hacia atrás" para cada elemento del vector x
- info: Indica el estado de salida de la rutina:
  - Si `info=0` : Ejecución con éxito
  - Si `info<0` : Si `info=-i` el i-ésimo argumento tuvo un valor ilegal.
  - Si `info>0` : Si `info=i`, y `i` es
    - <=N : U es exactamente cero. La factorización ha sido completada, pero el factor U es singular.
    - =N+1 : `rcond` es menor que la precisión de la maquina

Consideramos que son muchos los parametros y las opciones de ejecución en estas rutinas para expertos. A continuación mostramos un ejemplo que resuelve el mismo problema que la rutina pvgesv vista en el apartado anterior.

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvPOSVX"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
        b=ones((n,nrhs))
        print "b'=",transpose(b)
else:
        a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
a,af,equed,sr,sc,b,x,rcond,ferr,berr,info= PySLK.pvposvx(a,b)

x_num=PyACTS2Num(x)
if PyACTS.iread==1:
        print "a*x=b --> x'=",transpose(x_num)
        print "r=",transpose(r)
        print "c=",transpose(c)
        print "ferr:",transpose(ferr),"; berr=",transpose(berr)
        print "Info:",info
PyACTS.gridexit()
```

El resultado en la ejecución de este ejemplo sería el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvposvx.py
Ejemplo de Utilizacion ScaLAPACK: PvPOSVX
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 32 * 32
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [[1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [[ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]
 [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
sr= [ [ 0.  0.  0.  0.  0.  0.  0.  0.]]
sc= [ [ 0.  0.  0.  0.  0.  0.  0.  0.]]
ferr: [ [  2.10942375e-15   2.10942375e-15   0.00000000e+00   0.00000000e+00
          0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]] ;
berr= [ [  5.55111512e-17   5.55111512e-17   0.00000000e+00   0.00000000e+00
          0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]]
Info: 0
```

9.5. Rutinas especializadas para problemas de valores únicos en matrices generales y simétricas

9.5.1. pvsyevx

```
m,nz,w,z,ifail,iclustr,gap,info= PySLK.pvsyevx(a[,jobz,range,uplo,orfac,rcond,vl,vu,il,
```

La rutina 'pvsyevx' calcula los valores únicos seleccionados y, opcionalmente los vectores ortogonales de una matriz simétrica real. Los valores únicos y vectores ortogonales puedes ser seleccionados indicando un rango de valores o un conjunto de índices para los deseados valores únicos.

Las características de cada uno de los parámetros son las siguientes:

- Parámetros de Entrada
  - a: Matriz simétrica de dimensiones $n \times n$.
  - jobz: Indica si la rutina ha de calcular o no los vectores únicos.:
    - jobz='N': Calcula únicamente los valores únicos.
    - jobz='V': Calcula los valores únicos y los vectores únicos.
  - range: Indica qué valores únicos deberemos calcular::
    - range='A': Todos los valores únicos.
    - range='V': Todos los valores únicos en el intervalo [vl,vu].
    - range='I': Los valores únicos desde el il-ésimo hasta el iu-ésimo.
  - uplo: Especifica la parte de la matriz simétrica a que es utilizada para realizar el cálculo:
    - uplo='U': Únicamente la parte superior triangular de la matriz a es utilizada para definir los elementos de la matriz.
    - uplo='L': Únicamente la parte inferior triangular de la matriz a es utilizada para definir los elementos de la matriz.
  - vl: Si range='V', el limite inferior para buscar los valores únicos. Por defecto este valor es 0.
  - vu: Si range='V', el limite superior para buscar los valores únicos. Por defecto este valor es 0.
  - il: Si range='I', el índice inferior(desde el más pequeño hasta el más grande) de los valores únicos devueltos por defecto, il>=1.
  - iu: Si range='I', el índice superior (desde el más pequeño hasta el más grande) de los valores únicos devueltos por defecto, iu<=n.
  - orfac: Real. Indica qué vectores ortogonales deberían ser reortogonalizados.
- Parámetros de Salida
  - m: Número de valores únicos encontrados
  - nz: Número de vectores ortogonales computados.
  - w: Vector de tipo **Numeric** de tamaño n donde se devuelven los valores singulares de la matriz a.
  - z: Matriz de tipo **PyACTS** de tamaño $n \times n$ donde se devuelven los vectores singulares de la matriz a.
  - ifail: Vector de tipo **Numeric** de tamaño n. Si jobz='V', entonces los primeros m elemementos de ifail=0. Si (mod(info,2)<>0) entonces ifail contiene los indices del vector que falló en la convergencia.
  - icluster: Vector de tipo **Numeric** de tamaño $2 * nprow * npcol$. Este vector contiene los indices de los vectores ortogonales correspondientes a un cluster de valores únicos que no puedieron ser ortogonalizados debido a insuficiente espacio de trabajo (ver lwork, orfac y info).
  - gap: Vector de tipo **Numeric** de tamaño $nprow * npcol$. Este vector contiene el salto entre aquellos valores unicos que no pudieron ser ortogonalizados.

- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entonces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal, entonces `info=-i`.
  - `>0`: Si `mod(info,2)<>0`, entonces uno o mas vectores ortogonales fallaron en la convergencia.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvSYEVX"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
else:
        a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
m,nz,w,z,work,ifail,iclustr,gap,info= PySLK.pvsyevx(a)
z_num=PyACTS2Num(z)
if PyACTS.iread==1:
        print "Eigenvalues --> w'=",transpose(w)
        print "Eigenvectors --> z=",z_num
        print "Info:",info
PyACTS.gridexit()
```

De esta rutina queremos destacar el hecho que `w` es una matriz de tipo `Numeric` y será conocida por todos los procesos que intervienen en la malla. Sin embargo, z es una matriz de tipo `PyACTS`, lo que implica que sus valores se encuentran distribuidos entre los distintos procesos que intervienen en la malla. En el caso que quisieramos obtener toda los valores en un único proceso, éste debería llamar a la función `PyACTS2Num` tal y como se indica en este ejemplo. El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvsyevx.py
Ejemplo de Utilizacion ScaLAPACK: PvSYEVX
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 32 * 32
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Eigenvalues --> w'= [ 8.  8.  8.  8.  8.  8.  8.  8.]
Eigenvectors --> z= [[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
Info: 0
```

9.5.2. pvsygvx

`m,nz,w,z,ifail,iclustr,gap,info= PySLK.pvsyevx(a[,jobz,range,uplo,orfac,rcond,vl,vu,il,`

La rutina '`pvsygvx`' calcula todos los valores únicos y, opcionalmente los vectores ortogonales de un problema de valores únicos de la forma

$$A \times X = \lambda \times B \times X$$

,

$$A \times B \times X = \lambda \times X$$

o

$$B \times A \times X = \lambda \times X$$

Se asume que A es una matriz simetrica y B es una matriz simétrica positiva.

Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz simétrica de dimensiones $n \times n$.
  - `b`: Matriz simétrica de dimensiones $n \times n$.
  - `ibtype`: Número entero que indica el tipo de problema a ser resuelto:
    - `ibtype=1`: $A \times X = \lambda \times B \times X$
    - `ibtype=2`: $A \times B \times X = \lambda \times X$
    - `ibtype=3`: $B \times A \times X = \lambda \times X$
  - `range`: Indica qué valores únicos deberemos calcular::
    - `range='A'`: Todos los valores únicos.
    - `range='V'`: Todos los valores únicos en el intervalo [`vl`,`vu`].

    - range='I': Los valores únicos desde el il-ésimo hasta el iu-ésimo.
  - uplo: Especifica la parte de la matriz simétrica a que es utilizada para realizar el cálculo:
    - uplo='U': Únicamente la parte superior triangular de la matriz a es utilizada para definir los elementos de la matriz.
    - uplo='L': Únicamente la parte inferior triangular de la matriz a es utilizada para definir los elementos de la matriz.
  - vl: Si range='V', el limite inferior para buscar los valores únicos. Por defecto este valor es 0.
  - vu: Si range='V', el limite superior para buscar los valores únicos. Por defecto este valor es 0.
  - il: Si range='I', el índice inferior(desde el más pequeño hasta el más grande) de los valores únicos devueltos por defecto, il>=1.
  - iu: Si range='I', el índice superior (desde el más pequeño hasta el más grande) de los valores únicos devueltos por defecto, iu<=n.
  - abstol: Real. Indica el valor absoluto de tolerancia para los valores únicos.
  - orfac: Real. Indica qué vectores ortogonales deberían ser reortogonalizados.

- Parámetros de Salida
  - m: Número de valores únicos encontrados.
  - nz: Número de vectores ortogonales computados.
  - w: Vector de tipo **Numeric** de tamaño n donde se devuelven los valores singulares de la matriz a.
  - z: Matriz de tipo **PyACTS** de tamaño $n \times n$ donde se devuelven los vectores singulares de la matriz a.
  - ifail: Vector de tipo **Numeric** de tamaño n. Si jobz='V', entonces los primeros m elemementos de ifail=0. Si (mod(info,2)<>0) entonces ifail contiene los indices del vector que falló en la convergencia.
  - icluster: Vector de tipo **Numeric** de tamaño $2 * nprow * npcol$. Este vector contiene los indices de los vectores ortogonales correspondientes a un cluster de valores únicos que no puedieron ser ortogonalizados debido a insuficiente espacio de trabajo (ver lwork, orfac y info).
  - gap: Vector de tipo **Numeric** de tamaño $nprow * npcol$. Este vector contiene el salto entre aquellos valores unicos que no pudieron ser ortogonalizados.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entonces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal, entonces info=-i.
    - >0: Si mod(info,2)<>0, entonces uno o mas vectores ortogonales fallaron en la convergencia.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n=8
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
        print "Ejemplo de Utilizacion ScaLAPACK: PvSYEVX"
        print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
        print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
        a=8*identity(n,Float)
        print "a=",a
        b=0.5 * identity(n,Float)
        print "b=",b

else:
        a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
m,nz,w,z,ifail,iclustr,gap,info= PySLK.pvsygvx(a,b,1)
z_num=PyACTS2Num(z)
if PyACTS.iread==1:
        print "Eigenvalues --> w'=",transpose(w)
        print "Info:",info
PyACTS.gridexit()
```

De esta rutina queremos destacar el hecho que `w` es una matriz de tipo `Numeric` y será conocida por todos los procesos que intervienen en la malla. Sin embargo, z es una matriz de tipo `PyACTS`, lo que implica que sus valores se encuentran distribuidos entre los distintos procesos que intervienen en la malla. En el caso que quisieramos obtener toda los valores en un único proceso, éste debería llamar a la función `PyACTS2Num` tal y como se indica en este ejemplo. El resultado de este código es el siguiente:

```
[vgaliano@localhost EXAMPLES]$ mpirun -np 4 mpipython exPyScapvsygvx.py
Ejemplo de Utilizacion ScaLAPACK: PvSYEVX
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 32 * 32
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b= [[ 0.5  0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.5  0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.5  0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.5  0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.5  0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.5  0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.5]]
Eigenvalues --> w'= [ 16.  16.  16.  16.  16.  16.  16.  16.]
Info: 0
```

9.6. Rutinas computacionales para Ecuaciones Lineales

9.6.1. pvgetrf

`a,info= PySLK.pvgetrf(a[,ia=1,ja=1])`

La rutina '`pvgetrf`' obtiene la factorización LU de una matriz A de tamaño $m \times n$ distribuida utilizando una pivotación parcial con intercambio de filas.

La factorización tiene la forma $A = PLU$, donde P es una matriz permutación, L es una matriz triangular inferior con los elementos a uno en la diagonal principal y los elementos inferiores a la diagonal indicados en el resultado de la matriz A. U es una matriz triangular superior con sus elementos de la diagonal igual a uno y los elementos superiores en la parte de la diagonal superior de A.

La resolución de un sistema de ecuaciones se lleva a cabo mediante la descomposición LU utilizando la pivotacion parcial y el intercambio de filas de A. De este modo obtenemos `sub( A ) = P * L * U`, donde `P` es una matriz permutada, `L` es una matriz triangular unitaria, y `U` es una matriz triangular superior.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $A = PLU$.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito

- `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
- `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit()
if PyACTS.iread==1:
  print "Ejemplo de Utilizacion ScaLAPACK: PvGETRF"
  print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
  print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
  a=8*identity(n,Float)+ones([n,n],Float)
  print "a=",a
else:
  a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,info= PySLK.pvgetrf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
  print "P * L * U, =",a
  print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvgetrf.py
Ejemplo de Utilizacion ScaLAPACK: PvGETRF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 64 * 64
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
P * L * U, = [[ 9.          1.          1.          1.          1.          1.
        1.          1.        ]
 [ 0.11111111  8.88888889  0.88888889  0.88888889  0.88888889  0.88888889
        0.88888889  0.88888889]
 [ 0.11111111  0.1         8.8         0.8         0.8         0.8
        0.8         0.8       ]
 [ 0.11111111  0.1         0.09090909  8.72727273  0.72727273  0.72727273
        0.72727273  0.72727273]
 [ 0.11111111  0.1         0.09090909  0.08333333  8.66666667  0.66666667
        0.66666667  0.66666667]
 [ 0.11111111  0.1         0.09090909  0.08333333  0.07692308  8.61538462
        0.61538462  0.61538462]
 [ 0.11111111  0.1         0.09090909  0.08333333  0.07692308  0.07142857
        8.57142857  0.57142857]
 [ 0.11111111  0.1         0.09090909  0.08333333  0.07692308  0.07142857
        0.06666667  8.53333333]]
Info: 0
```

9.6.2. pvgetrs

`b,info= PySLK.pvgetrs(a,b[,ia=1,ja=1,ib=1,jb=1])`

La rutina '`pvgetrs`' resuelve la ecuación del sistema lineal siguiente:

$$AX = B$$

donde A es una matriz $N \times N$ que utiliza la factorización LU implementada en `pvgetrf`.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes de dimensiones $n \times n$.
  - `b`: Vector de términos independiente de dimensiones $n \times nrhs$.
- `Parámetros de Salida`
  - `b`: Vector distribuido donde se almacena el resultado.
  - `info`: Resultado global de la ejecución
    - `o = 0`: Ejecución con éxito

- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGETRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)
    print "a=",a
    b=ones((n,nrhs))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
a,info= PySLK.pvgetrf(a)
b,info= PySLK.pvgetrs(a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
    print "a*x=b --> x'=",transpose(b_num)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0: mpirun -np 2 mpipython exPyScapvgetrs.py
Ejemplo de Utilizacion ScaLAPACK: PvGETRS
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [ [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [ [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
Info: 0
```

9.6.3. pvgecon

`anorm,rcond,info=pvgecon(ACTS_a,[norm='O',ia=1,ja=1])`

La rutina '`pvgecon`' estima el recíproco del numero condición de una matriz destribuida A, mediante la norma 1 o mediante la norma infinito, utilizando la factorización LU computada mediante pvgetrf:

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes de dimensiones $n \times n$.
  - `norm`: Parámetro opcional que indica la norma a calcular.
    - `norm='1' o norm='0'`: Norma 1
    - `norm='I'`: Norma infinita
- `Parámetros de Salida`
  - `anorm`: Norma resultante
  - `rcond`: Recíproco del número de condición
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGECON"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,nrhs))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
anorm,rcond,info= PySLK.pvgecon(a)

if PyACTS.iread==1:
    print "anorm:",anorm
    print "rcond:",rcond
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvgecon.py
Ejemplo de Utilizacion ScaLAPACK: PvGECON
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1 1]]
anorm: 0.0
rcond: 0.0
Info: 0
```

9.6.4. pvgecon

`anorm,rcond,info=pvgecon(ACTS_a,[norm='O',ia=1,ja=1])`

La rutina '`pvgecon`' estima el recíproco del numero condición de una matriz distribuida A, mediante la norma 1 o mediante la norma infinito, utilizando la factorización LU computada mediante pvgetrf:

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes de dimensiones $n \times n$.
  - `norm`: Parámetro opcional que indica la norma a calcular.
    - `norm='1'` o `norm='0'`: Norma 1
    - `norm='I'`: Norma infinita
- `Parámetros de Salida`
  - `anorm`: Norma resultante
  - `rcond`: Recíproco del número de condición
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGECON"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,nrhs))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call PBLAS routine
anorm,rcond,info= PySLK.pvgecon(a)

if PyACTS.iread==1:
    print "anorm:",anorm
    print "rcond:",rcond
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvgecon.py
Ejemplo de Utilizacion ScaLAPACK: PvGECON
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1 1]]
anorm: 0.0
rcond: 0.0
Info: 0
```

9.6.5. pvgerfs

`x,ferr,berr,info= PySLK.pvgerfs(a,af,b,x,[trans='N',ia=1,ja=1,ib=1,jb=1,ix=1,jx=1,iaf=1,`

La rutina '`pvgerfs`' implementa la solución a un sistema de ecuaciones lineal y proporciona estimación de error para las soluciones aportadas.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes de dimensiones $n \times n$.
  - `af`: Matriz factorizada en los términos $A = PLU$ mediante la rutina pvgetrf de dimensiones $n \times n$.
  - `b`: Matriz de términos independientes $n \times nrhs$.
  - `x`: Matriz de aproximación a la solución de dimensiones $n \times nrhs$.
  - `trans`: Parámetro opcional que indica si se realiza la transformada.
    - `trans='N'`: No realiza la traspuesta $AX = B$
    - `trans='T'`: Se realiza la traspuesta $A^T X = B$
    - `trans='C'`: Se realiza la traspuesta conjugada $A^T X = B$
- `Parámetros de Salida`
  - `x`: Vector resultado
  - `ferr`: Estimación de error para adelante
  - `berr`: Estimación de error hacia atrás
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGERFS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    af=8*identity(n,Float)+ones([n,n],Float)
    b=ones((n,nrhs))
    print "b'=",transpose(b)
    x=ones((n,nrhs))
    print "x'=",transpose(x)
else:
    a,af,b,x=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
af=Num2PyACTS(af,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
#We call PBLAS routine
af,info= PySLK.pvgetrf(af)
x,ferr,berr,info= PySLK.pvgerfs(a,af,b,x)
x=PyACTS2Num(x)
if PyACTS.iread==1:
    print "Solución x:",x
    print "ferr:",ferr
    print "berr:",berr
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 vgaliano@nodo0$ mpirun -np 2 mpipython exPyScapvgerfs.py
Ejemplo de Utilizacion ScaLAPACK: PvGERFS
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1 1]]
x'= [ [1 1 1 1 1 1 1 1]]
Solution x: [[ 0.0625]
 [ 0.0625]
 [ 0.0625]
 [ 0.0625]
 [ 0.0625]
 [ 0.0625]
 [ 0.0625]]
ferr: [[  9.92261828e-15]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]]
berr: [[  9.43689571e-16]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]]
Info: 0
```

9.6.6. pvgetri

```
a, info = PySLK.pvgetri(a,[ia=1,ja=1])
```

La rutina '`pvgetri`' computa la inversa de una matriz distribuida mediante la factorización LU computada mediante pvgetrf. Este método invierte la matriz U y entonces computa la inversa de A denotada como A^{-1} mediante la resolución del sistema $A^{-1}L = U^{-1}$.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`

- `a`: Matriz inversa
- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGETRI"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,info= PySLK.pvgetri(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "Inverse :",a
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0$ mpirun -np 4 mpipython exPyScapvgetri.py
Ejemplo de Utilizacion ScaLAPACK: PvGETRI
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
Inverse : [[ -1.37174211e-03  -1.21932632e-03  -1.08384562e-03  -9.63418329e-04
        -8.56371848e-04  -7.61219420e-04  -6.08975536e-03   1.23456790e-01]
 [  1.23456790e-01  -1.37174211e-03  -1.21932632e-03  -1.08384562e-03
        -9.63418329e-04  -8.56371848e-04  -6.85097478e-03  -1.11111111e-01]
 [ -1.11111111e-01   1.23456790e-01  -1.37174211e-03  -1.21932632e-03
        -1.08384562e-03  -9.63418329e-04  -7.70734663e-03   0.00000000e+00]
 [  0.00000000e+00  -1.11111111e-01   1.23456790e-01  -1.37174211e-03
        -1.21932632e-03  -1.08384562e-03  -8.67076496e-03   0.00000000e+00]
 [ -1.38777878e-17   0.00000000e+00  -1.11111111e-01   1.23456790e-01
        -1.37174211e-03  -1.21932632e-03  -9.75461058e-03   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00  -1.11111111e-01
         1.23456790e-01  -1.37174211e-03  -1.09739369e-02   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
        -1.11111111e-01   1.23456790e-01  -1.23456790e-02   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
         0.00000000e+00  -1.11111111e-01   1.11111111e-01   0.00000000e+00]]
Info: 0
```

9.6.7. pvgeequ

`r,c,rowcnd,colcnd,amax,info= PySLK.pvgeequ(a,[ia=1,ja=1])`

La rutina '`pvgeequ`' computa la ponderación de filas y columna para equilibrar la matriz distribuida A y reducir su numero de condición. R devuelve los factores de escala de filas y C devuelve los factores de escala de columnas. Éstos son elegidos para hacer más grandes las entradas en cada fila y columna de la matriz distribuida B para que sus elementos elementos $Bij = RiAijCj$ tengan valor absoluto igual a 1.

Ri y Cj son valores restringidos entre `s`mlnum y `b`ignum. El uso de estos factores de escalado no esta garantizado para lograr una reducción del número de condición A pero en práctica el resultado suele ser bueno.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `r`:Vector local que contienen los factores de escala para las filas
  - `c`:Vector local que contienen los factores de escala para las columnas
  - `rowcnd`: si `i`nfo=0 contiene el ratio de Ri más pequeño al más grande Ri.

- `colcnd`: si `info=0` contiene el ratio de Ci más pequeño al más grande Ci.
- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGEEQU"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
r,c,rowcnd,colcnd,amax,info= PySLK.pvgeequ(a)
if PyACTS.iread==1:
    print "r :",r
    print "c :",c
    print "rowcnd :",rowcnd
    print "colcnd :",colcnd
    print "amax :",amax
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0$ mpirun -np 4 mpipython exPyScapvgequ.py
Ejemplo de Utilizacion ScaLAPACK: PvGEEQU
N= 8 ;nprow x npcol: 2 x 2
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
r : [[ 0.11111111]
 [ 0.11111111]
 [ 0.11111111]
 [ 0.11111111]]
c : [[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]
rowcnd : 1.0
colcnd : 1.0
amax : 9.0
Info: 0
```

9.6.8. pvgbtrf

```
a= PySLK.pvgbtrf(ACTS_a,[bwl,bwu,ja])
```

La rutina '`pvgbtrf`' computa una factorización LU de un sistema $N \times N$ real en bandas con anchos de las bandas `bwu` y `bwl`. La reordenación se utiliza para incrementar el paralelismo en la factorización. Esta reordenación produce que los factores sean diferentes a los obtenidos en codigos secuenciales. Estos factores no pueden ser utilizados directamente por los usuarios. Sin embargo, pueden ser utilizados en llamadas a pvgbtrs para solucionar sistemas lineales. La factorización tiene el siguiente aspecto:

$$PAQ = LU$$

donde U es una matriz triangular con banda superior, L es una matriz triangular con banda inferior, y L y Q son matrices de permutación. La matriz Q representa la reordenación de las columnas para implementar el paralelismo, mientras que P representa la matriz de reordenación de filas para implementar la estabilidad numérica que utiliza la pivotación clásica parcial.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `bwu`: (opcional) Ancho de banda superior, por defecto `bwu=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
  - `bwl`: (opcional) Ancho de banda inferior, por defecto `bwl=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
- `Parámetros de Salida`
  - `a`:Contiene las matriz factorizada.

- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGBTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 1=Scalapack
bw=2
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw)
#We call ScaLAPACK routine
print "bwu=",bw,";bwl=",bw
a,info= PySLK.pvgbtrf(a,bwl=bw,bwu=bw)

a=PyACTS2Num(a,bwu=bw,bwl=bw)
if PyACTS.iread==1:
    print "a=",a
    print "Info=",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0$ mpirun -np 1 exPyScapvgbtrf.py
Ejemplo de Utilizacion ScaLAPACK: PvGBTRF
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
bwu= 2 ;bwl= 2
a= [[ 9.          1.          1.          0.          0.          0.
        0.        ]
 [ 0.11111111  8.88888889  0.88888889  1.          0.          0.
        0.        ]
 [ 0.11111111  0.1         8.8         0.9         1.          0.
        0.        ]
 [ 0.          0.1125      0.10227273  8.79545455  0.89772727  1.
        0.        ]
 [ 0.          0.          0.11363636  0.89772727  8.88636364  1.
        1.        ]
 [ 0.          0.          0.          1.          1.          9.
        1.        ]
 [ 0.          0.          0.          0.          1.          1.
        9.        ]]
Info = 0
```

9.6.9. pvgbtrs

`b,info= PySLK.pvgbtrs(a,b[,bwl=bw,bwu=bw)]`

La rutina '`pvgbtrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

o

$$A^T X = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por pvgbtrf. A es una matriz $N \times N$ distribuidas por bandas mediante bwu y bwl. La rutina pvgbtrf debe ser llamada previamente.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- Parámetros de Entrada
  - a: Matriz de coeficientes de dimensiones $n \times n$.
  - b: Matriz de términos independientes dimensiones $n \times 1$.
  - bwu: (opcional) Ancho de banda superior, por defecto bwu=n-1 pero se puede especificar cualquier valor entre 0 y n-1.
  - bwl: (opcional) Ancho de banda inferior, por defecto bwl=n-1 pero se puede especificar cualquier valor entre 0 y n-1.
- Parámetros de Salida

- b:Solución al sistema de ecuaciones.
- info: Resultado global de la ejecución
  - = 0: Ejecución con éxito
  - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
  - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGBTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 1=Scalapack
bw=2
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw)
#We call ScaLAPACK routine
print "bwu=",bw,";bwl=",bw
a,info= PySLK.pvgbtrf(a,bwl=bw,bwu=bw)

a=PyACTS2Num(a,bwu=bw,bwl=bw)
if PyACTS.iread==1:
    print "a=",a
    print "Info=",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 1 /home/vgaliano/mpipython/mpipython exPyScapvgbtrs.py
Ejemplo de Utilizacion ScaLAPACK: PvGBTRS
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 8 * 8
a= [[ 8.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.]]
b'= [ [1 1 1 1 1 1 1]]
bwu= 2 ;bwl= 2
0
```

9.6.10. pvdbtrf

```
a= PySLK.pvdbtrf(ACTS_a,[bwl,bwu,ja])
```

La rutina '`pvdbtrf`' computa una factorización LU de un sistema $N \times N$ real en bandas con anchos de las bandas `bwu` y `bwl`. La reordenación se utiliza para incrementar el paralelismo en la factorización. Esta reordenación produce que los factores sean diferentes a los obtenidos en codigos secuenciales. Estos factores no pueden ser utilizados directamente por los usuarios. Sin embargo, pueden ser utilizados en llamadas a pvdbtrs para solucionar sistemas lineales. La factorización tiene el siguiente aspecto:

$$PAP^T = LU$$

donde U es una matriz triangular con banda superior, L es una matriz triangular con banda inferior, y L y Q son matrices de permutación. La matriz Q representa la reordenación de las columnas para implementar el paralelismo, mientras que P representa la matriz de reordenación de filas para implementar la estabilidad numérica que utiliza la pivotación clásica parcial. Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `bwu`: (opcional) Ancho de banda superior, por defecto `bwu=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
  - `bwl`: (opcional) Ancho de banda inferior, por defecto `bwl=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
- `Parámetros de Salida`
  - `a`:Contiene las matriz factorizada.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvDBTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='db'
bw=2
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw,piv=pivot)
#We call ScaLAPACK routine
print "bwu=",bw,";bwl=",bw
a,info= PySLK.pvdbtrf(a,bwl=bw,bwu=bw)
a=PyACTS2Num(a,bwu=bw,bwl=bw,piv=pivot)
if PyACTS.iread==1:
    print "a=",a
    print "info=",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0$ mpirun -np 1 exPyScapvdbtrf.py
Ejemplo de Utilizacion ScaLAPACK: PvDBTRF
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
bwu= 2 ;bwl= 2
a= [[ 9.          1.          1.          0.          0.          0.
        0.        ]
 [ 0.11111111  8.88888889  0.88888889  1.          0.          0.
        0.        ]
 [ 0.11111111  0.1         8.8         0.9         1.          0.
        0.        ]
 [ 0.          0.1125      0.10227273  8.79545455  0.89772727  1.
        0.        ]
 [ 0.          0.          0.11363636  0.10206718  8.79473514  0.89793282
        1.        ]
 [ 0.          0.          0.          0.11369509  0.1020989   8.79462695
        0.8979011 ]
 [ 0.          0.          0.          0.          0.11370439  0.10209655
        8.794623  ]]
info= 0

Info = 0
```

9.6.11. pvdbtrs

`b,info= PySLK.pvdbtrs(a,b[,bwl=bw,bwu=bw)]`

La rutina '`pvgbtrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

o

$$A^T X = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por pvgbtrf. A es una matriz $N \times N$ distribuidas por bandas mediante bwu y bwl. La rutina pvgbtrf debe ser llamada previamente.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- Parámetros de Entrada
  - a: Matriz de coeficientes de dimensiones $n \times n$.
  - b: Matriz de términos independientes dimensiones $n \times 1$.
  - bwu: (opcional) Ancho de banda superior, por defecto bwu=n-1 pero se puede especificar cualquier valor entre 0 y n-1.
  - bwl: (opcional) Ancho de banda inferior, por defecto bwl=n-1 pero se puede especificar cualquier valor entre 0 y n-1.

- Parámetros de Salida
  - b:Solución al sistema de ecuaciones.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
    - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvDBTRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,1))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='db'
bw=2
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw,piv=pivot)
b=Num2PyACTS(b,502)
#We call ScaLAPACK routine
print "bwu=",bw,";bwl=",bw
a,info= PySLK.pvdbtrf(a,bwl=bw,bwu=bw)
b,info= PySLK.pvdbtrs(a,b,bwl=bw,bwu=bw)
if PyACTS.iread==1:
    print "Info:",info
    print "B:",b.data
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 1 mpipython exPyScapvdbtrs.py
Ejemplo de Utilizacion ScaLAPACK: PvDBTRS
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1]]
bwu= 2 ;bwl= 2
Info: 0
B: [ 0.09348148  0.084       0.07466667  0.07585185  0.07466667  0.084
       0.09348148]
```

9.6.12. pvdttrf

```
ACTS_ad,info= PySLK.pvdttrf(ACTS_ad)
```

La rutina '`pvdbtrf`' computa una factorización LU de un sistema tridiagonal $N \times N$ real. La reordenación se utiliza para incrementar el paralelismo en la factorización. Esta reordenación produce que los factores sean diferentes a los obtenidos en codigos secuenciales. Estos factores no pueden ser utilizados directamente por los usuarios. Sin embargo, pueden ser utilizados en llamadas a pvdttrs para solucionar sistemas lineales. La factorización tiene el siguiente aspecto:

$$PAP^T = LU$$

donde U es una matriz triangular con banda superior, L es una matriz triangular con banda inferior, y L y Q son matrices de permutación. La matriz Q representa la reordenación de las columnas para implementar el paralelismo, mientras que P representa la matriz de reordenación de filas para implementar la estabilidad numérica que utiliza la pivotación clásica parcial.

Esta rutina tiene una particularidad, y es que la matriz a se convierte en tres vectores que identifican la banda inferior `dl`, la diagonal principal `d`, y la banda superior `du`. De este modo, cuando se ejecuta `Num2PyACTS`, la propiedad `a.data` realmente contiene un conjunto de estos tres vectores. Este manejo de variables es transparente al usuario y se puede apreciar con el ejemplo indicado.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a tipo PyACTS.
- `Parámetros de Salida`
  - `a`:Contiene el conjunto de las tres diagonales.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvDTTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='dt'
ACTS_ad=Num2PyACTS(a,ACTS_lib,piv=pivot)
#We call ScaLAPACK routine
a,info= PySLK.pvdttrf(ACTS_ad)
if PyACTS.iread==1:
    print "a=",a.data
    print "info=",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
a= [array([ 1.        ,  0.11111111,  0.1125    ,  0.11251758,  0.1125178 ,  0.11251781]), a
            9.        ]), array([ 1.,  1.,  1.,  1.,  1.,  1.])]
info= 0
```

9.6.13. pvdttrs

`b,info= PySLK.pvdttrs(ACTS_ad,b)` La rutina '`pvdttrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

o

$$A^T X = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por `pvdttrf`. A es una matriz $N \times N$ distribuidas por bandas mediante `bwu` y `bwl`. La rutina `pvdttrs` debe ser llamada previamente.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a tipo PyACTS.
  - `b`: Vector de términos independientes convertido a tipo 502 (Right Hand) .
- `Parámetros de Salida`
  - `b`: Vector solución.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvDTTRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,1))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='dt'
ACTS_ad=Num2PyACTS(a,ACTS_lib,piv=pivot)
b=Num2PyACTS(b,502)
#We call ScaLAPACK routine
ACTS_ad,info= PySLK.pvdttrf(ACTS_ad)
b,info= PySLK.pvdttrs(ACTS_ad,b)
if PyACTS.iread==1:
    print "Info:",info
    print "B:",b.data
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 1 mpipython exPyScapvdttrs.py
Ejemplo de Utilizacion ScaLAPACK: PvDTTRS
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1]]
Info: 0
B: [ 0.10113772  0.08976056  0.09101722  0.09108443  0.08922292  0.10590925
      -0.04240616]
```

9.6.14. pvpotrf

`a,info= PySLK.pvpotrf(a,uplo=Ü")`

La rutina '`pvpotrf`' computa la factorización Cholesky de una matriz simétrica definida positiva. La factorización tiene el siguiente aspecto:

$$A = U^T U, si \texttt{ uplo='U'}$$

o

$$A = L^T L, si \texttt{ uplo='L'}$$

donde U es una matriz triangular superior, y L es una matriz triangular inferior. Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz simétrica definida positiva previamente convertida a tipo PyACTS.
- `Parámetros de Salida`
  - `a`:Contiene la factorización realizada.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPOTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,info= PySLK.pvpotrf(a,uplo="U")
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "U^T * U =",a
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvpotrf.py
Ejemplo de Utilizacion ScaLAPACK: PvPOTRF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
U^T * U = [[ 3.          0.33333333  0.33333333  0.33333333  0.33333333  0.33333333
        0.33333333  0.33333333]
 [ 1.          2.98142397  0.2981424   0.2981424   0.2981424   0.2981424
        0.2981424   0.2981424 ]
 [ 1.          1.          2.96647939  0.26967994  0.26967994  0.26967994
        0.26967994  0.26967994]
 [ 1.          1.          1.          2.95419578  0.24618298  0.24618298
        0.24618298  0.24618298]
 [ 1.          1.          1.          1.          2.94392029  0.22645541
        0.22645541  0.22645541]
 [ 1.          1.          1.          1.          1.          2.93519754
        0.20965697  0.20965697]
 [ 1.          1.          1.          1.          1.          1.
        2.92770022  0.19518001]
 [ 1.          1.          1.          1.          1.          1.
        1.          2.92118697]]
Info: 0
```

9.6.15. pvpocon

`rcond,info= PySLK.pvpocon(a,uplo='U',anorm=1)` La rutina '`pvpocon`' estima el numero condición (con norma 1) de una matriz simétrica definida positiva utilizando la factorización Cholesky $A = U^T U$ o $A = LL^T$ computada mediante el método pspotrf.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a
  - `anorm`: Número que indica la norma a aplicar. Por defecto, el valor es 1.
  - `uplo`: Indica si los factores son almacenados en la parte superior o inferior de la matriz. tipo PyACTS.
  - `b`: Vector de términos independientes.
- `Parámetros de Salida`
  - `rcond`: Número de condición.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.

- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPOCON"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,info= PySLK.pvpotrf(a,uplo="U")
rcond,info= PySLK.pvpocon(a,uplo='U',anorm=1)

if PyACTS.iread==1:
    print "Condition Number =",rcond
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0:mpirun -np 2 mpipython exPyScapvpocon.py
Ejemplo de Utilizacion ScaLAPACK: PvPOCON
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
Condition Number = 5.81818181818
Info: 0
```

9.6.16. pvporfs

x,ferr,berr,info= PySLK.pvporfs(a,af,b,x,[uplo=Ü"]) La rutina 'pvporfs' calcula la solución a un sistema lineal de ecuaciones cuando la matriz de coeficientes es simétrica definida positiva y proporciona una

estimación del error en la solución.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz simétrica definida positiva de dimensiones $n \times n$.
  - `af`: Matriz tridiagonal de dimensiones $n \times n$.
  - `b`: Matriz tridiagonal de dimensiones $n \times n$.
  - `x`: Matriz tridiagonal de dimensiones $n \times n$.
  - `uplo`: Indica si los factores son almacenados en la parte superior o inferior de la matriz. tipo PyACTS.
  - `b`: Vector de términos independientes.
- `Parámetros de Salida`
  - `rcond`: Número de condición.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPORFS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)
    af=8*identity(n,Float)
    print "a=",a
    b=ones((n,nrhs))
    x=ones((n,nrhs))
    print "b'=",transpose(b)
else:
    a,af,b,x=None,None,None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
af=Num2PyACTS(af,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
x=Num2PyACTS(x,ACTS_lib)
#We call PBLAS routine
x,ferr,berr,info= PySLK.pvporfs(a,af,b,x,uplo="U")
if PyACTS.iread==1:
    print "Ferr=",transpose(ferr)
    print "Berr=",transpose(berr)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
Ejemplo de Utilizacion ScaLAPACK: PvPORFS
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [ [1 1 1 1 1 1 1 1]]
Ferr= [ [ 0.10745614  0.          0.          0.          0.          0.
          0.          0.        ]]
Berr= [ [ 0.75384615  0.          0.          0.          0.          0.
          0.          0.        ]]
Info: 0
```

9.6.17. pvpotri

`a, info = PySLK.pvpotri(a,[uplo='U',ia=1,ja=1])`

La rutina '`pvgetri`' computa la inversa de una matriz distribuida simétrica definida positiva mediante la factorización de Cholesky $A = U^T U$ o $A = LL^T$ computada mediante pvpotrf.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `uplo`: Indica si los factores son almacenados en la parte superior o inferior de la matriz.
- `Parámetros de Salida`
  - `a`: Matriz inversa
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPOTRI"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,info= PySLK.pvpotrf(a,uplo="U")
a,info= PySLK.pvpotri(a,uplo="U")
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A^-1 =",a
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvpotri.py
Ejemplo de Utilizacion ScaLAPACK: PvPOTRI
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A^-1 = [[ 0.1171875 -0.0078125 -0.0078125 -0.0078125 -0.0078125 -0.0078125 -0.0078125
       -0.0078125]
 [ 1.         0.1171875 -0.0078125 -0.0078125 -0.0078125 -0.0078125 -0.0078125
       -0.0078125]
 [ 1.         1.         0.1171875 -0.0078125 -0.0078125 -0.0078125 -0.0078125
       -0.0078125]
 [ 1.         1.         1.         0.1171875 -0.0078125 -0.0078125 -0.0078125
       -0.0078125]
 [ 1.         1.         1.         1.         0.1171875 -0.0078125 -0.0078125
       -0.0078125]
 [ 1.         1.         1.         1.         1.         0.1171875 -0.0078125
       -0.0078125]
 [ 1.         1.         1.         1.         1.         1.         0.1171875
       -0.0078125]
 [ 1.         1.         1.         1.         1.         1.         1.
        0.1171875]]
Info: 0
```

9.6.18. pvpoequ

```
sr,sc,rowcnd,colcnd,amax,info= PySLK.pvgeequ(a,[ia=1,ja=1])
```

La rutina '`pvpoequ`' computa la ponderación de filas y columna para equilibrar la matriz distribuida A simétrica definida positiva y reducir su numero de condición. SR y SC contienen los factores de escala $Si = 1/\sqrt{Aii}$, elegidos para que los elementos de la matriz B escalada sea $Bij = SiAijSj$ tenga unos en la diagonal. Esta elección de SR y SC establece el numero de condición de B con el valor más pequeño sobre la diagonal.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `sr`:Vector local que contienen los factores de escala para las filas
  - `sc`:Vector local que contienen los factores de escala para las columnas
  - `scond`: si `info`=0 contiene el ratio de SRi más pequeño al más grande SRi.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito

- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPOEQU"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
sr,sc,scond,amax,info= PySLK.pvpoequ(a)
if PyACTS.iread==1:
    print "sr :",transpose(sr)
    print "sc :",transpose(sc)
    print "scond :",transpose(scond)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 2 mpipython exPyScapvpoequ.py
Ejemplo de Utilizacion ScaLAPACK: PvPOEQU
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
sr : [ [ 0.33333333  0.33333333  0.33333333  0.33333333]]
sc : [ [ 0.33333333  0.33333333  0.33333333  0.33333333  0.33333333  0.33333333
         0.33333333  0.33333333]]
scond : 1.0
Info: 0
```

9.6.19. pvpbtrf

```
a,info= PySLK.pvpbtrf(ACTS_a,[bw,ja])
```

La rutina '`pvpbtrf`' computa una factorización de Cholesky de una matriz simétrica definida positiva $N \times N$ en bandas con ancho de la banda `bw`. La reordenación se utiliza para incrementar el paralelismo en la factorización. Esta reordenación produce que los factores sean diferentes a los obtenidos en códigos secuenciales. Estos factores no pueden ser utilizados directamente por los usuarios. Sin embargo, pueden ser utilizados en llamadas a pvdbtrs para solucionar sistemas lineales. La factorización tiene el siguiente aspecto:

$$PAP^T = U'U, \; si \; \texttt{uplo} =' U'$$

o

$$PAP^T = LL', \; si \; \texttt{uplo} =' L'$$

donde U es una matriz triangular con banda superior, L es una matriz triangular con banda inferior, y P es una matriz de permutación.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `bw`: (opcional) Ancho de banda superior, por defecto `bwu=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
- `Parámetros de Salida`
  - `a`:Contiene las matriz factorizada.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPBTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 1=Scalapack
bw=2
pivot='pb'
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw,piv=pivot)
#We call ScaLAPACK routine
print "bwu=",bw,";bwl=",bw
a,info= PySLK.pvpbtrf(a,bw=2)
a=PyACTS2Num(a,bwu=bw,bwl=bw,piv=pivot)
if PyACTS.iread==1:
    print "A=",a
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 1 /home/vgaliano/mpipython/mpipython exPyScapvpbtrf.py
Ejemplo de Utilizacion ScaLAPACK: PvPBTRF
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
bwu= 2 ;bwl= 2
A= [[ 3.          0.33333333  0.33333333  0.          0.          0.
        0.        ]
 [ 1.          2.98142397  0.2981424   0.3354102   0.          0.
        0.        ]
 [ 1.          1.          2.96647939  0.30338994  0.33709993  0.
        0.        ]
 [ 0.          1.          1.          2.96571316  0.30270199  0.33718703
        0.        ]
 [ 0.          0.          1.          1.          2.96559187  0.30278368
        0.33720082]
 [ 0.          0.          0.          1.          1.          2.96557363
        0.30277485]
 [ 0.          0.          0.          0.          1.          1.
        2.96557296]]
Info: 0
```

9.6.20. pvdbtrs

`b,info= PySLK.pvpbtrs(a,b[,bw=i)]`

La rutina '`pvpbtrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

o

$$A^T X = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por `pvpbtrf`. A es una matriz $N \times N$ distribuidas por bandas mediante `bwu` y `bwl`. La rutina `pvpbtrf` debe ser llamada previamente.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de coeficientes de dimensiones $n \times n$.
  - `b`: Matriz de términos independientes dimensiones $n \times 1$.
  - `bw`: (opcional) Ancho de banda superior, por defecto `bw=n-1` pero se puede especificar cualquier valor entre 0 y n-1.
- `Parámetros de Salida`
  - `b`:Solución al sistema de ecuaciones.

- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPBTRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,1))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='pb'
bw=2
a=Num2PyACTS(a,ACTS_lib,bwu=bw,bwl=bw,piv=pivot)
b=Num2PyACTS(b,502)
#We call ScaLAPACK routine
print "bw=",bw
a,info= PySLK.pvpbtrf(a,bw=bw)
b,info= PySLK.pvpbtrs(a,b,bw=bw)
if PyACTS.iread==1:
    print "Info:",info
    print "B:",b.data
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 1 mpipython exPyScapvpbtrs.py
Ejemplo de Utilizacion ScaLAPACK: PvPBTRS
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1]]
bwu= 2 ;bwl= 2
Info: 0
B: [ 0.09348148  0.084       0.07466667  0.07585185  0.07466667  0.084
       0.09348148]
```

9.6.21. pvpttrf

```
ACTS_ad,info= PySLK.pvpttrf(ACTS_ad)
```

La rutina '`pvdbtrf`' computa una factorización LU de un sistema tridiagonal simétrico $N \times N$. La reordenación se utiliza para incrementar el paralelismo en la factorización. Esta reordenación produce que los factores sean diferentes a los obtenidos en codigos secuenciales. Estos factores no pueden ser utilizados directamente por los usuarios. Sin embargo, pueden ser utilizados en llamadas a pvdttrs para solucionar sistemas lineales. La factorización tiene el siguiente aspecto:

$$PAP^T = U'DU$$

o

$$PAP^T = LDL'$$

donde U es una matriz triangular con banda superior, L es una matriz triangular con banda inferior, y P es una matriz de permutación.

Esta rutina tiene una particularidad, y es que la matriz a se convierte en tres vectores que identifican la banda inferior `dl`, la diagonal principal `d`, y la banda superior `du`. De este modo, cuando se ejecuta `Num2PyACTS`, la propiedad `a.data` realmente contiene un conjunto de estos tres vectores. Este manejo de variables es transparente al usuario y se puede apreciar con el ejemplo indicado.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a tipo PyACTS. Su propiedad `data` contiene dos vectores (`d` y `e`) que
- `Parámetros de Salida`
  - `a`:Contiene el conjunto de las tres diagonales.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.

- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPTTRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='pt'
ACTS_ad=Num2PyACTS(a,ACTS_lib,piv=pivot)
#We call ScaLAPACK routine
a,info= PySLK.pvpttrf(ACTS_ad)
if PyACTS.iread==1:
    print "a=",a.data
    print "info=",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 1 mpipython exPyScapvpttrf.py
Ejemplo de Utilizacion ScaLAPACK: PvDTTRF
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
[ 9.  9.  9.  9.  9.  9.  9.] [ 1.  1.  1.  1.  1.  1.]
a= [array([ 9.        ,  8.88888889,  8.8875    ,  8.88748242,  8.8874822 ,  8.88748219,
            8.88748219]), array([ 0.11111111,  0.1125    ,  0.11251758,  0.1125178 ,  0.112
info= 0
```

9.6.22. pvpttrs

`b,info= PySLK.pvpttrs(ACTS_ad,b)`
La rutina '`pvpttrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por `pvpttrf`. A es una matriz $N \times N$ distribuidas por bandas mediante `bwu` y `bwl`. La rutina `pvpttrf` debe ser llamada previamente.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a tipo PyACTS. Su propiedad `data` contiene dos vectores (`d` y `e`) que representan los elementos de la diagonal principal y de la banda contigua.
  - `b`: Vector de términos independientes convertido a tipo 502 (Right Hand) .
- `Parámetros de Salida`
  - `b`: Vector solución.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *

n=7
#Initiliaze the Grid
PyACTS.gridinit(nb=4,nprow=1)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvPTTRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
    b=ones((n,1))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=501 # 501=Scalapack; Partial pivoting Band
pivot='pt'
ACTS_ad=Num2PyACTS(a,ACTS_lib,piv=pivot)
b=Num2PyACTS(b,502)
#We call ScaLAPACK routine
ACTS_ad,info= PySLK.pvpttrf(ACTS_ad)
b,info= PySLK.pvpttrs(ACTS_ad,b)
if PyACTS.iread==1:
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 1 /home/vgaliano/mpipython/mpipython exPyScapvpttrs.py
Ejemplo de Utilizacion ScaLAPACK: PvPTTRS
N= 7 ;nprow x npcol: 1 x 1
Tam. Bloques: 4 * 4
a= [[ 9.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.]]
b'= [ [1 1 1 1 1 1 1]]
Info: 0
```

9.6.23. pvtrtrs

`b,info= PySLK.pvtrtrs(ACTS_ad,b)`
La rutina '`pvpttrs`' resuelve un sistema de ecuaciones lineal del tipo:

$$AX = B$$

donde A es una matriz utilizada para producir los factores almacenados en A y AF por pvpttrf. A es una matriz $N \times N$ distribuida triangular. Previamente se debe haber comprobado que A es una matriz no singular.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- Parámetros de Entrada
  - a: Matriz tridiagonal de dimensiones $n \times n$, previamente convertida a tipo PyACTS. Su propiedad data contiene dos vectores (d y e) que representan los elementos de la diagonal principal y de la banda contigua.
  - b: Vector de términos independientes convertido.
- Parámetros de Salida
  - b: Vector solución.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
    - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,1
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGETRS"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)
    print "a=",a
    b=ones((n,nrhs))
    print "b'=",transpose(b)
else:
    a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call PBLAS routine
b,info= PySLK.pvtrtrs(a,b)
b_num=PyACTS2Num(b)
if PyACTS.iread==1:
    print "a*x=b --> x'=",transpose(b_num)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
vgaliano@nodo0:mpirun -np 2 mpipython exPyScapvtrtrs.py
Ejemplo de Utilizacion ScaLAPACK: PvGETRS
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
b'= [ [1 1 1 1 1 1 1 1]]
a*x=b --> x'= [ [ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]]
Info: 0
```

9.7. Rutinas computacionales para Factorizaciones ortogonales

9.7.1. pvgeqpf

`a,tau,info= PySLK.pvgeqpf(a[,ia=1,ja=1])`
La rutina '`pvgeqpf`' obtiene una factorización QR con pivotación de una matriz A de tamaño $m \times n$ distribuida utilizando una pivotación parcial con intercambio de filas.

$$AP = QR$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz R, los elementos inferiores a la diagonal contienen a la matriz Q.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGEQPF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqpf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A*P =Q*R-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvgeqpf.py
Ejemplo de Utilizacion ScaLAPACK: PvGEQPF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A*P =Q*R--> [[-9.38083152 -2.5584086  -2.5584086  -2.5584086  -2.5584086  -2.5584086
       -2.5584086  -2.5584086 ]
 [ 0.0544045  -9.0252172  -1.93397511 -1.93397511 -1.93397511 -1.93397511
       -1.93397511 -1.93397511]
 [ 0.0544045   0.04522339 -8.81557064 -1.55568894 -1.55568894 -1.55568894
       -1.55568894 -1.55568894]
 [ 0.0544045   0.04522339  0.03900305 -8.67721831 -1.30158275 -1.30158275
       -1.30158275 -1.30158275]
 [ 0.0544045   0.04522339  0.03900305  0.03445855 -8.57904424 -1.11900577
       -1.11900577 -1.11900577]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769 -8.50575253
       -0.98143298 -0.98143298]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769  0.02818813
       -8.44894167 -0.87402845]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769  0.02818813
        0.02591422  8.4036117 ]]
Tau: [ [ 1.95940322  1.97575564  1.98490246  1.99054577  1.99426252  1.99682676
        1.99865781  0.        ]]
Info: 0
```

9.7.2. pvgeqrf

`a,tau,info= PySLK.pvgeqrf(a[,ia=1,ja=1])`
La rutina '`pvgeqrf`' obtiene una factorización QR con pivotación de una matriz A de tamaño $m \times n$ distribuida utilizando una pivotación parcial con intercambio de filas.

$$A = QR$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz R, los elementos inferiores a la diagonal contienen a la matriz Q.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito

- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGEQRF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqrf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A =Q*R-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
  mpirun -np 2 mpipython exPyScapvgeqrf.py
Ejemplo de Utilizacion ScaLAPACK: PvGEQRF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A =Q*R--> [[-9.38083152 -2.5584086  -2.5584086  -2.5584086  -2.5584086  -2.5584086
       -2.5584086  -2.5584086 ]
 [ 0.0544045  -9.0252172  -1.93397511 -1.93397511 -1.93397511 -1.93397511
       -1.93397511 -1.93397511]
 [ 0.0544045   0.04522339 -8.81557064 -1.55568894 -1.55568894 -1.55568894
       -1.55568894 -1.55568894]
 [ 0.0544045   0.04522339  0.03900305 -8.67721831 -1.30158275 -1.30158275
       -1.30158275 -1.30158275]
 [ 0.0544045   0.04522339  0.03900305  0.03445855 -8.57904424 -1.11900577
       -1.11900577 -1.11900577]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769 -8.50575253
       -0.98143298 -0.98143298]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769  0.02818813
       -8.44894167 -0.87402845]
 [ 0.0544045   0.04522339  0.03900305  0.03445855  0.03096769  0.02818813
        0.02591422  8.4036117 ]]
Tau: [ [ 1.95940322  1.97575564  1.98490246  1.99054577  1.99426252  1.99682676
        1.99865781  0.        ]]
Info: 0
```

9.7.3. pvorgqr

`a,tau,info= PySLK.pvorgqr(a[k=0,ia=1,ja=1])`
La rutina '`pvorgrf`' genera una matriz Q de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = H1H2 \ldots H3$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
  - `info`: Resultado global de la ejecución

- = 0: Ejecución con éxito
- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvORGQR"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqrf(a)
a,tau,info= PySLK.pvorgqr(a,k=1)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A =Q*R-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvorgqr.py
Ejemplo de Utilizacion ScaLAPACK: PvORGQR
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A =Q*R--> [[-0.95940322 -0.10660036 -0.10660036 -0.10660036 -0.10660036 -0.10660036
       -0.10660036 -0.10660036]
 [-0.10660036  0.99420046 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954  0.99420046 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954  0.99420046 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954  0.99420046 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954  0.99420046
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
        0.99420046 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954  0.99420046]]
Tau: [ [ 1.95940322  1.97575564  1.98490246  1.99054577  1.99426252  1.99682676
         1.99865781  0.        ]]
Info: 0
```

9.7.4. pvormqr

`c,info= PySLK.pvormqr(a[k=0,side='L',trans='N',ia=1,ja=1])`
La rutina '`pvormqr`' sobreescribe la matriz distribuida C por :

Si `trans`='N': QC (si `side`='L') o CQ (si `side`='R')
Si `trans`='T': Q^TC (si `side`='R') o CQ^T (si `side`='L')

donde Q es una matriz de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = H1H2 \dots H3$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.
  - `c`: Matriz de dimensiones $n \times n$.

- `tau`: Obtenido de la ejecución de pvgeqrf.
- `side`: Indica si la operación se realiza por la izquierda (side='L') o derecha (side='R').
- `trans`: Indica si la operación se realiza con la matriz o con su traspuesta.
- `k`: Número de elementos reflectores cuyo producto define la matriz Q. Si side='L', entonces $M \geq k \geq 0$, si side='R', entonces $N \geq k \geq 0$.

■ `Parámetros de Salida`

- `a`: Matriz Q que contiene los vectores ortonormales.
- `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORMQR"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=c=",a
c=8*identity(n,Float)
else:
a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqrf(a)
c,info= PySLK.pvormqr(a,c,tau,k=1,side='R',trans='T')
c=PyACTS2Num(c)
if PyACTS.iread==1:
print "C-->",c
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvormqr.py
Ejemplo de Utilizacion ScaLAPACK: PvORMQR
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[-7.67522579 -0.85280287 -0.85280287 -0.85280287 -0.85280287 -0.85280287
       -0.85280287 -0.85280287]
 [-0.85280287  7.95360368 -0.04639632 -0.04639632 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632  7.95360368 -0.04639632 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632  7.95360368 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632  7.95360368 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632  7.95360368
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632 -0.04639632
        7.95360368 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632 -0.04639632
       -0.04639632  7.95360368]]
Info: 0
```

9.7.5. pvgelqf

`a,tau,info= PySLK.pvgelqf(a[,ia=1,ja=1])`
La rutina '`pvgelqf`' obtiene una factorización LQ con pivotación de una matriz A de tamaño $m \times n$ distribuida :

$$A = LQ$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $A = LQ$. La parte triangular superior contiene la matriz Q, los elementos inferiores a la diagonal contienen a la matriz L.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.

- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGELQF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgelqf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A*P =Q*R-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvgelqf.py
Ejemplo de Utilizacion ScaLAPACK: PvGELQF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A*P =Q*R--> [[-9.38083152  0.0544045   0.0544045   0.0544045   0.0544045   0.0544045
        0.0544045   0.0544045 ]
 [-2.5584086  -9.0252172   0.04522339  0.04522339  0.04522339  0.04522339
        0.04522339  0.04522339]
 [-2.5584086  -1.93397511 -8.81557064  0.03900305  0.03900305  0.03900305
        0.03900305  0.03900305]
 [-2.5584086  -1.93397511 -1.55568894 -8.67721831  0.03445855  0.03445855
        0.03445855  0.03445855]
 [-2.5584086  -1.93397511 -1.55568894 -1.30158275 -8.57904424  0.03096769
        0.03096769  0.03096769]
 [-2.5584086  -1.93397511 -1.55568894 -1.30158275 -1.11900577 -8.50575253
        0.02818813  0.02818813]
 [-2.5584086  -1.93397511 -1.55568894 -1.30158275 -1.11900577 -0.98143298
       -8.44894167  0.02591422]
 [-2.5584086  -1.93397511 -1.55568894 -1.30158275 -1.11900577 -0.98143298
       -0.87402845  8.4036117 ]]
Tau: [ [ 1.95940322  1.97575564  1.99426252  1.99682676  0.          0.
        0.          0.        ]]
Info: 0
```

9.7.6. pvorglq

`a,tau,info= PySLK.pvorglq(a[k=0,ia=1,ja=1])`
La rutina '`pvorglq`' genera una matriz Q de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = H1H2 \ldots H3$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `tau`: Obtenido de la ejecución de pvgelq.
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Obtenido de la ejecución de pvgelq.

- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORGLQ"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgelqf(a)
a,tau,info= PySLK.pvorglq(a,tau,k=1)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print "A =L*Q-->",a
print "Tau:",transpose(tau)
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
$ mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvorglq.py
Ejemplo de Utilizacion ScaLAPACK: PvORGLQ
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A =L*Q--> [[-0.95940322 -0.10660036 -0.10660036 -0.10660036 -0.10660036 -0.10660036
       -0.10660036 -0.10660036]
 [-0.10660036  0.99420046 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954  0.99420046 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954  0.99420046 -0.00579954 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954  0.99420046 -0.00579954
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954  0.99420046
       -0.00579954 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
        0.99420046 -0.00579954]
 [-0.10660036 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954  0.99420046]]
Tau: [ [ 1.95940322  1.97575564  1.99426252  1.99682676  0.          0.
        0.          0.        ]]
Info: 0
vgaliano
```

9.7.7. pvormlq

```
c,info= PySLK.pvormlq(a[k=0,side='L',trans='N',ia=1,ja=1])
```

La rutina '`pvormlq`' sobreescribe la matriz distribuida C por :

Si `trans`='N': QC (si `side`='L') o CQ (si `side`='R')
Si `trans`='T': Q^TC (si `side`='R') o CQ^T (si `side`='L')

donde Q es una matriz de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = Hk \ldots H2H1$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.

- `c`: Matriz de dimensiones $n \times n$.
- `tau`: Obtenido de la ejecución de pvgelq.
- `side`: Indica si la operación se realiza por la izquierda (`side='L'`) o derecha (`side='R'`).
- `trans`: Indica si la operación se realiza con la matriz o con su traspuesta.
- `k`: Número de elementos reflectores cuyo producto define la matriz Q. Si `side='L'`, entonces $M \geq k \geq 0$, si `side='R'`, entonces $N \geq k \geq 0$.

▪ `Parámetros de Salida`

- `a`: Matriz Q que contiene los vectores ortonormales.
- `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORMLQ"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=c=",a
c=8*identity(n,Float)
else:
a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgelqf(a)
c,info= PySLK.pvormlq(a,c,tau,k=1,side='R',trans='T')
c=PyACTS2Num(c)
if PyACTS.iread==1:
print "C-->",c
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvormlq.py
Ejemplo de Utilizacion ScaLAPACK: PvORMLQ
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[-7.67522579 -0.85280287 -0.85280287 -0.85280287 -0.85280287 -0.85280287
       -0.85280287 -0.85280287]
 [-0.85280287  7.95360368 -0.04639632 -0.04639632 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632  7.95360368 -0.04639632 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632  7.95360368 -0.04639632 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632  7.95360368 -0.04639632
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632  7.95360368
       -0.04639632 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632 -0.04639632
        7.95360368 -0.04639632]
 [-0.85280287 -0.04639632 -0.04639632 -0.04639632 -0.04639632 -0.04639632
       -0.04639632  7.95360368]]
Info: 0
```

9.7.8. pvgeqlf

`a,tau,info= PySLK.pvgeqlf(a[,ia=1,ja=1])`
La rutina '`pvgeqlf`' obtiene una factorización LQ con pivotación de una matriz A de tamaño $m \times n$ distribuida :

$$A = QL$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $A = LQ$. La parte triangular superior contiene la matriz Q, los elementos inferiores a la diagonal contienen a la matriz L.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.

- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvGEQLF"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqlf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "A=L*Q-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 2 mpipython exPyScapvgeqlf.py
Ejemplo de Utilizacion ScaLAPACK: PvGEQLF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A =L*Q--> [[ 8.4036117   0.02591422  0.02818813  0.03096769  0.03445855  0.03900305
        0.04522339  0.0544045 ]
 [-0.87402845 -8.44894167  0.02818813  0.03096769  0.03445855  0.03900305
        0.04522339  0.0544045 ]
 [-0.98143298 -0.98143298 -8.50575253  0.03096769  0.03445855  0.03900305
        0.04522339  0.0544045 ]
 [-1.11900577 -1.11900577 -1.11900577 -8.57904424  0.03445855  0.03900305
        0.04522339  0.0544045 ]
 [-1.30158275 -1.30158275 -1.30158275 -1.30158275 -8.67721831  0.03900305
        0.04522339  0.0544045 ]
 [-1.55568894 -1.55568894 -1.55568894 -1.55568894 -1.55568894 -8.81557064
        0.04522339  0.0544045 ]
 [-1.93397511 -1.93397511 -1.93397511 -1.93397511 -1.93397511 -1.93397511
       -9.0252172   0.0544045 ]
 [-2.5584086  -2.5584086  -2.5584086  -2.5584086  -2.5584086  -2.5584086
       -2.5584086  -9.38083152]]
Tau: [ [ 0.          1.99865781  1.99682676  1.99426252  1.99054577  1.98490246
        1.97575564  1.95940322]]
Info: 0
```

9.7.9. pvorgql

`a,tau,info= PySLK.pvorglq(a[k=0,ia=1,ja=1])`
La rutina '`pvorgql`' genera una matriz Q de tamaño $m \times n$ con columnas ortonormales, las cuales están definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = Hk \ldots H2, H1$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `tau`: Vector obtenido en `p`vgeqlf
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Vector que contiene los factores escalares de los vectores ortnormales.

- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORGQL"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgeqlf(a)
a,tau,info= PySLK.pvorgql(a,tau,k=1)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print "A =Q*L-->",a
print "Tau:",transpose(tau)
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvorgql.py
Ejemplo de Utilizacion ScaLAPACK: PvORGQL
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A =Q*L--> [[ 0.99420046 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.10660036]
 [-0.00579954  0.99420046 -0.00579954 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.10660036]
 [-0.00579954 -0.00579954  0.99420046 -0.00579954 -0.00579954 -0.00579954
       -0.00579954 -0.10660036]
 [-0.00579954 -0.00579954 -0.00579954  0.99420046 -0.00579954 -0.00579954
       -0.00579954 -0.10660036]
 [-0.00579954 -0.00579954 -0.00579954 -0.00579954  0.99420046 -0.00579954
       -0.00579954 -0.10660036]
 [-0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954  0.99420046
       -0.00579954 -0.10660036]
 [-0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954 -0.00579954
        0.99420046 -0.10660036]
 [-0.10660036 -0.10660036 -0.10660036 -0.10660036 -0.10660036 -0.10660036
       -0.10660036 -0.95940322]]
Tau: [ [ 0.          1.99865781  1.99682676  1.99426252  1.99054577  1.98490246
        1.97575564  1.95940322]]
Info: 0
```

9.7.10. pvormql

`c,info= PySLK.pvormql(a[k=0,side='L',trans='N',ia=1,ja=1])`
La rutina '`pvormql`' sobreescribe la matriz distribuida C por :
Si `trans`='N': QC (si `side`='L') o CQ (si `side`='R')
Si `trans`='T': Q^TC (si `side`='R') o CQ^T (si `side`='L')
donde Q es una matriz de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = Hk \dots H2, H1$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.
  - `tau`: Vector obtenido en `pvgeqlf`
  - `c`: Matriz de dimensiones $n \times n$.
  - `side`: Indica si la operación se realiza por la izquierda (`side`='L') o derecha (`side`='R').
  - `trans`: Indica si la operación se realiza con la matriz o con su traspuesta.

- k: Número de elementos reflectores cuyo producto define la matriz Q. Si side='L', entonces $M \geq k \geq 0$, si side='R', entonces $N \geq k \geq 0$.

- Parámetros de Salida

  - a: Matriz Q que contiene los vectores ortonormales.
  - tau: Vector que contiene los factores escalares de los vectores ortnormales.
  - info: Resultado global de la ejecución
    - = 0: Ejecución con éxito
    - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
    - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvORMQL"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=c=",a
    c=8*identity(n,Float)
else:
    a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
c,tau,info= PySLK.pvgeqlf(c)
c,info= PySLK.pvormql(a,c,tau,k=1,side='R',trans='T')
c=PyACTS2Num(c)
if PyACTS.iread==1:
    print "C-->",c
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvormql.py
Ejemplo de Utilizacion ScaLAPACK: PvORMQL
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Info: 0
```

9.7.11. pvgerqf

`a,tau,info= PySLK.pvgerqf(a[,ia=1,ja=1])`
La rutina '`pvgeqlf`' obtiene una factorización RQ con pivotación de una matriz A de tamaño $m \times n$ distribuida :

$$A = RQ$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $A = RQ$. La parte triangular superior contiene la matriz Q, los elementos inferiores a la diagonal contienen a la matriz L.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvGERQF"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgerqf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print "A=R*Q-->",a
print "Tau:",transpose(tau)
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvgerqf.py
Ejemplo de Utilizacion ScaLAPACK: PvGERQF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
A=R*Q--> [[ 8.4036117  -0.87402845 -0.98143298 -1.11900577 -1.30158275 -1.55568894
       -1.93397511 -2.5584086 ]
 [ 0.02591422 -8.44894167 -0.98143298 -1.11900577 -1.30158275 -1.55568894
       -1.93397511 -2.5584086 ]
 [ 0.02818813  0.02818813 -8.50575253 -1.11900577 -1.30158275 -1.55568894
       -1.93397511 -2.5584086 ]
 [ 0.03096769  0.03096769  0.03096769 -8.57904424 -1.30158275 -1.55568894
       -1.93397511 -2.5584086 ]
 [ 0.03445855  0.03445855  0.03445855  0.03445855 -8.67721831 -1.55568894
       -1.93397511 -2.5584086 ]
 [ 0.03900305  0.03900305  0.03900305  0.03900305  0.03900305 -8.81557064
       -1.93397511 -2.5584086 ]
 [ 0.04522339  0.04522339  0.04522339  0.04522339  0.04522339  0.04522339
       -9.0252172  -2.5584086 ]
 [ 0.0544045   0.0544045   0.0544045   0.0544045   0.0544045   0.0544045
        0.0544045  -9.38083152]]
Tau: [ [ 0.          1.99865781  1.99054577  1.98490246  0.          0.
        0.          0.        ]]
Info: 0
```

9.7.12. pvorgrq

`a,tau,info= PySLK.pvorgrq(a,tau[k=0,ia=1,ja=1])`
La rutina '`pvorgql`' genera una matriz Q de tamaño $m \times n$ con columnas ortonormales, las cuales están definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = Hk \ldots H2, H1$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
  - `info`: Resultado global de la ejecución

- = 0: Ejecución con éxito
- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORGRQ"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgerqf(a)
a,tau,info= PySLK.pvorgrq(a,tau,k=1)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print "A =R*Q-->",a
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 /home/vgaliano/mpipython/mpipython exPyScapvormrq.py
Ejemplo de Utilizacion ScaLAPACK: PvORMRQ
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Info: 0
```

9.7.13. pvormrq

`c,info= PySLK.pvormrq(a,c,tau,[k=0,side='L',trans='N',ia=1,ja=1])`
La rutina '`pvormrq`' sobreescribe la matriz distribuida C por :
Si `trans`='N': QC (si `side`='L') o CQ (si `side`='R')
Si `trans`='T': Q^TC (si `side`='R') o CQ^T (si `side`='L')
donde Q es una matriz de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = H1, H2, \ldots Hk$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.
  - `c`: Matriz de dimensiones $n \times n$.
  - `tau`: Vector obtenido mediante `pvgerqf`.
  - `side`: Indica si la operación se realiza por la izquierda (`side`='L') o derecha (`side`='R').
  - `trans`: Indica si la operación se realiza con la matriz o con su traspuesta.
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q. Si `side`='L', entonces $M \geq k \geq 0$, si `side`='R', entonces $N \geq k \geq 0$.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
  - `info`: Resultado global de la ejecución

- = 0: Ejecución con éxito
- <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
- <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORMRQ"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=c=",a
c=8*identity(n,Float)
else:
a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgerqf(a)
c,info= PySLK.pvormrq(a,c,tau,k=1,side='R',trans='T')
c=PyACTS2Num(c)
if PyACTS.iread==1:
print "C-->",c
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvormrq.py
Ejemplo de Utilizacion ScaLAPACK: PvORMRQ
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Info: 0
```

9.7.14. pvtzrzf

`a,tau,info= PySLK.pvtzrzf(a[,ia=1,ja=1])`
La rutina '`pvtzrzf`' reduce la $m \times n$ parte superior de una matriz real a la forma que implica su transformada ortogonal.

$$A = R0Z,$$

donde Z es una matriz $N \times N$ ortogonal y R es una matriz $M \times M$ triangular superior. Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $A = RQ$. La parte triangular superior contiene la matriz Q, los elementos inferiores a la diagonal contienen a la matriz L.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvTZRZF"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvtzrzf(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print "( R  0 ) * Z -->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
 mpirun -np 2 mpipython exPyScapvtzrzf.py
Ejemplo de Utilizacion ScaLAPACK: PvTZRZF
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
( R  0 ) * Z --> [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
Tau: [ [ 0.  0.  0.  0.  0.  0.  0.  0.]]
Info: 0
```

9.7.15. pvormrz

`c,info= PySLK.pvormrz(a,c,tau,[k=0,l=0,side='L',trans='N'])`
La rutina '`pvormrq`' sobreescribe la matriz distribuida C por :
Si `trans`='N': QC (si `side`='L') o CQ (si `side`='R')
Si `trans`='T': Q^TC (si `side`='R') o CQ^T (si `side`='L')
donde Q es una matriz de tamaño $m \times n$ con columnas ortonormales, las cuales estan definidas como las primeras n columnas de un producto de k reflectores elementales de orden m

$$Q = H1, H2, \ldots Hk$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.
  - `c`: Matriz de dimensiones $n \times n$.
  - `tau`: Vector obtenido mediante `pvtzrzf`.
  - `side`: Indica si la operación se realiza por la izquierda (`side`='L') o derecha (`side`='R').
  - `trans`: Indica si la operación se realiza con la matriz o con su traspuesta.
  - `k`: Número de elementos reflectores cuyo producto define la matriz Q. Si `side`='L', entonces $M \geq k \geq 0$, si `side`='R', entonces $N \geq k \geq 0$.
- `Parámetros de Salida`
  - `a`: Matriz Q que contiene los vectores ortonormales.
  - `tau`: Vector que contiene los factores escalares de los vectores ortnormales.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvORMRZ"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=c=",a
    c=8*identity(n,Float)
else:
    a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvtzrzf(a)
c,info= PySLK.pvormrz(a,c,tau,k=1,l=1,side='R',trans='T')
c=PyACTS2Num(c)
if PyACTS.iread==1:
    print "C-->",c
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 1 /home/vgaliano/mpipython/mpipython exPyScapvormrz.py
Ejemplo de Utilizacion ScaLAPACK: PvORMRZ
N= 8 ;nprow x npcol: 1 x 1
Tam. Bloques: 2 * 2
a=c= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
C--> [[ 8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  8.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  8.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.]]
Info: 0
```

9.8. Rutinas computacionales para Problemas de valores propios en matrices simétricas

9.8.1. pvsytrd

`a,tau,d,e,info= PySLK.pvsytrd(a[uplo='U'])`
La rutina '`pvsytrd`' reduce una matriz simétrica A a una forma tridiagonal T simétrica mediante una transformación del siguiente tipo:

$$Q'AQ = T$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz R, los elementos inferiores a la diagonal contienen a la matriz Q.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
    print "Ejemplo de Utilizacion ScaLAPACK: PvSYTRD"
    print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
    print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
    a=8*identity(n,Float)+ones([n,n],Float)
    print "a=",a
else:
    a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvsytrd(a,uplo='U')
a=PyACTS2Num(a)
if PyACTS.iread==1:
    print " Q' * sub( A ) * Q = T-->",a
    print "Tau:",transpose(tau)
    print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 3 mpipython exPyScapvsytrd.py
Ejemplo de Utilizacion ScaLAPACK: PvSYTRD
N= 8 ;nprow x npcol: 3 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
 Q' * sub( A ) * Q = T-->
 [[  7.99999905e+00  -6.47092293e-07   4.14213568e-01   3.66025418e-01
          0.00000000e+00   0.00000000e+00   2.89897949e-01   2.74291903e-01]
 [  1.00000000e+00   7.99999952e+00   4.21468485e-07   3.66025418e-01
          0.00000000e+00   0.00000000e+00   2.89897949e-01   2.74291903e-01]
 [  1.00000000e+00   1.00000000e+00   8.00000000e+00  -3.09714807e-07
          0.00000000e+00   0.00000000e+00   2.89897949e-01   2.74291903e-01]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   8.00000000e+00
          0.00000000e+00   0.00000000e+00   2.89897949e-01   2.74291903e-01]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
          8.00000000e+00  -4.76837158e-07   2.89897949e-01   2.74291903e-01]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
          1.00000000e+00   7.99999619e+00   5.84003885e-07   2.74291903e-01]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
          1.00000000e+00   1.00000000e+00   1.50000000e+01  -2.64575124e+00]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
          1.00000000e+00   1.00000000e+00   1.00000000e+00   9.00000000e+00]]
Tau: [ [ 0.          0.          1.70710683  1.57735026  0.          0.
        1.40824831  1.3779645 ]]
Info: 0
```

9.8.2. pvormtr

`c,info= PySLK.pvormtr(a,c,tau[side='L',uplo='U',trans='N'])`
La rutina '`pvormtr`' sobreeescribe en la matriz $M \times N$ por :

	`side='L'`	`side='R'`
`trans='N'`	QC	CQ
`trans='T'`	Q^TC	CQ^T

donde Q es una matriz real ortogonal distribuida de orden nq, donde $nq = m$ si side='L' y $nq = n$ si side='R' . Q está definida como el producto de $nq - 1$ elementos reflectores devueltos por pvsytrd.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- Parámetros de Entrada
  - a: Matriz de dimensiones $m \times n$.
  - c: Matriz de dimensiones $m \times n$.
  - tau: Vector de elementos reflectores obtenido de pvsytrd.

- `side`: (Opcional) indica cómo ordenar los operadores
- `uplo`: (Opcional) Parte superior o inferior de la matriz A
- `trans`: (Opcional) Se realiza sobre la traspuesta de Q

- `Parámetros de Salida`
  - `a`: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz R, los elementos inferiores a la diagonal contienen a la matriz Q.
  - `tau`: Vector que contiene los factores escalares.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORMTR"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
c=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvsytrd(a,uplo='U')
c,info= PySLK.pvormtr(a,c,tau,side='L',uplo='U',trans='N')
c=PyACTS2Num(c)
if PyACTS.iread==1:
print " Q * sub( C )   =",c
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
mpirun -np 2 mpipython exPyScapvormtr.py
Ejemplo de Utilizacion ScaLAPACK: PvORMTR
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
 Q * sub( C )   =
 [[  5.64958374e+00  -2.17872498e+00   8.07120983e-01   2.76681884e+00
        -2.88117266e+00  -1.24169731e+00  -3.03098630e+00  -7.27051238e-03]
 [ -7.07833832e+00  -3.59293854e+00  -6.07092579e-01   1.35260528e+00
        -4.29538622e+00  -2.65591087e+00  -4.44519986e+00  -1.42148407e+00]
 [ -1.02445780e+00  -4.20968718e+00  -5.87146673e+00  -1.35520620e+00
         3.40100907e+00  -2.25888460e+00  -4.04817359e+00  -1.02445780e+00]
 [  7.96307252e-01   1.58399702e+00   6.54249961e+00   4.65558852e-01
         5.22177413e+00  -4.38119548e-01  -2.22740853e+00   7.96307252e-01]
 [ -1.72093663e+00  -1.59775765e+00  -1.76713419e+00  -8.43336633e+00
        -4.59483878e+00  -2.95536343e+00  -4.74465241e+00  -1.72093663e+00]
 [  1.84234827e-01   6.80150440e+00  -2.29753403e+00   2.00998262e+00
        -4.49924747e-02  -1.05019197e+00  -2.83948096e+00   1.84234827e-01]
 [  5.47855627e-01   5.47855627e-01   5.47855627e-01   5.47855627e-01
         5.47855627e-01   7.95441642e+00  -2.47586016e+00   5.47855627e-01]
 [  1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
         1.00000000e+00   1.00000000e+00   1.00000000e+00   9.00000000e+00]]
Info: 0
```

9.9. Rutinas computacionales para Problemas de valores propios en matrices no simétricas

9.9.1. pvgehrd

`a,tau,info= PySLK.pvgehrd(a[ilo=1,ihi=-1])`
La rutina '`pvgehrd`' reduce una matriz A a su forma Hessenberg superior H mediante una transformación ortogonal:

$$Q'AQ = H$$

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $n \times n$.
  - `ilo`: Índice inferior para los vectores ortonormales.
  - `ihi`: Índice superior para los vectores ortonormales.
- `Parámetros de Salida`

- a: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz de Hessenberg superior H, y los elementos inferiores a la diagonal contienen a la matriz Q junto con el vector tau.
- tau: Vector que contiene los factores escalares.
- info: Resultado global de la ejecución
  - = 0: Ejecución con éxito
  - <0: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces info=-(i*100+j). Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces info=-i.
  - <0: Si info=k. La factorización ha sido completada pero el factor U es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvGEHRD"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
print "a=",a
else:
a=None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgehrd(a)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print " Q' * sub( A ) * Q = T-->",a
print "Tau:",transpose(tau)
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

```
$ mpirun -np 2 mpipython exPyScapvgehrd.py

Ejemplo de Utilizacion ScaLAPACK: PvGEHRD
N= 8 ;nprow x npcol: 2 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
 Q' * sub( A ) * Q = T--> [[  9.00000000e+00  -2.64575131e+00   4.44089210e-16   0.00000000e
         0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [ -2.64575131e+00   1.50000000e+01   8.88178420e-16  -1.50323342e-15
        -2.61329217e-17  -1.20495759e-16  -1.96539658e-16  -1.08176581e-16]
 [  2.74291885e-01  -2.17558393e-15   8.00000000e+00   2.18463005e-15
         7.58594935e-17   3.49779000e-16   5.70521699e-16   3.14018492e-16]
 [  2.74291885e-01   2.89897949e-01   2.97904098e-15   8.00000000e+00
         8.88178420e-16   0.00000000e+00   1.77635684e-15   0.00000000e+00]
 [  2.74291885e-01   2.89897949e-01   3.09016994e-01   8.88178420e-16
         8.00000000e+00  -1.25607397e-15  -1.33226763e-15   0.00000000e+00]
 [  2.74291885e-01   2.89897949e-01   3.09016994e-01   3.33333333e-01
        -6.28036983e-16   8.00000000e+00  -1.57009246e-16   0.00000000e+00]
 [  2.74291885e-01   2.89897949e-01   3.09016994e-01   3.33333333e-01
         4.14213562e-01  -2.35513869e-16   8.00000000e+00   0.00000000e+00]
 [  2.74291885e-01   2.89897949e-01   3.09016994e-01  -3.33333333e-01
         0.00000000e+00   1.00000000e+00   0.00000000e+00   8.00000000e+00]]
Tau: [ [ 1.37796447  1.40824829  1.4472136   1.5         1.70710678  1.
         0.          0.        ]]
Info: 0
```

9.9.2. pvormhr

`c,info= PySLK.pvormhr(ACTS_a,ACTS_c,tau,ilo=1,ihi=-1,side='L',trans='N'])`
La rutina '`pvormhr`' sobreeescribe en la matriz $M \times N$ por :

	`side='L'`	`side='R'`
`trans='N'`	QC	CQ
`trans='T'`	Q^TC	CQ^T

donde Q es una matriz real ortogonal distribuida de orden nq, donde $nq = m$ si side='L' y $nq = n$ si side='R' . Q está definida como el producto de de `i`hi-ilo elementos reflectores devueltos por `pvgehrd`.

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz de dimensiones $m \times n$.
  - `c`: Matriz de dimensiones $m \times n$.
  - `tau`: Vector de elementos reflectores obtenido de `pvsytrd`.

- `ilo`: Índice inferior para los vectores ortonormales.
- `ihi`: Índice superior para los vectores ortonormales.
- `side`: (Opcional) indica cómo ordenar los operadores
- `uplo`: (Opcional) Parte superior o inferior de la matriz A
- `trans`: (Opcional) Se realiza sobre la traspuesta de Q

▪ `Parámetros de Salida`

- `a`: Forma factorizada de la matriz $AP = QR$. La parte triangular superior contiene la matriz R, los elementos inferiores a la diagonal contienen a la matriz Q.
- `tau`: Vector que contiene los factores escalares.
- `info`: Resultado global de la ejecución
  - `= 0`: Ejecución con éxito
  - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
  - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvORMHR"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
c=identity(n,Float)
print "a=",a
else:
a,c=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
c=Num2PyACTS(c,ACTS_lib)
#We call ScaLAPACK routine
a,tau,info= PySLK.pvgehrd(a)
c,info= PySLK.pvormhr(a,c,tau,side='L',trans='N')
c=PyACTS2Num(c)
if PyACTS.iread==1:
print " Q * sub( C )   =",c
print "Info:",info
PyACTS.gridexit()
```

El resultado de este código es el siguiente:

	Tipo	Factorización	Reducción	Obtención de vect. propios
1.	$Az = \lambda Bz$	$B = LL^T$ $B = U^T U$	$C = L^{-1} A L^{-T}$ $C = U^{-T} A U^{-1}$	$z = L^{-T} y$ $z = U^{-1} y$
2.	$ABz = \lambda z$	$B = LL^T$ $B = U^T U$	$C = L^T A L$ $C = U^T A U$	$z = L^{-T} y$ $z = U^{-1} y$
3.	$BAz = \lambda z$	$B = LL^T$ $B = U^T U$	$C = L^T A L$ $C = U A U^T$	$z = Ly$ $z = U^T y$

Cuadro 9.1: Reducción de problemas de valores propios simétricos generalizados a problemas estándar

```
vgaliano@nodo0:~mpirun -np 1 mpipython exPyScapvormhr.py
Ejemplo de Utilizacion ScaLAPACK: PvORMHR
N= 8 ;nprow x npcol: 1 x 1
Tam. Bloques: 2 * 2
a= [[ 9.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  9.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  9.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  9.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  9.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  9.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  9.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  9.]]
 Q * sub( C )   =
 [[  1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
          0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00  -3.77964473e-01   9.25820100e-01   2.86846478e-16
          0.00000000e+00  -3.82461971e-17   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01   9.12870929e-01
          4.29954161e-16  -2.05924331e-16   7.81734838e-17   3.90867419e-17]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01  -1.82574186e-01
          8.94427191e-01   1.54898152e-16  -4.30229898e-17   1.42614004e-17]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01  -1.82574186e-01
         -2.23606798e-01   8.66025404e-01   2.10531269e-16   5.38700218e-17]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01  -1.82574186e-01
         -2.23606798e-01  -2.88675135e-01   8.16496581e-01  -8.49078562e-17]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01  -1.82574186e-01
         -2.23606798e-01  -2.88675135e-01  -4.08248290e-01  -7.07106781e-01]
 [  0.00000000e+00  -3.77964473e-01  -1.54303350e-01  -1.82574186e-01
         -2.23606798e-01  -2.88675135e-01  -4.08248290e-01   7.07106781e-01]]
Info: 0
```

9.10. Rutinas computacionales para Problemas de valores propios en matrices simétricas definidas

9.10.1. pvsygst

```
a,tau,info= PySLK.pvsygst(a[ilo=1,ihi=-1])
```

La rutina '`pvsygst`' reduce una matriz A definida-simétrica a su forma estándar:

Esta rutina se provee para matrices con elementos de tipo real y complejo. Las características de cada uno de los

parámetros son las siguientes:

- `Parámetros de Entrada`
  - `a`: Matriz simétrica definida de dimensiones $n \times n$.
  - `b`: Matriz resultado de la factorización de cholesky al ejecutar previamente `pvpotrf`.
  - `ibtype`: Tipo de reducción utilizada a partir de la tabla 9.1.
  - `uplo`: Parte de la matriz utilizada para los cálculos: superior `uplo='U'` o inferior `uplo='L'`.
- `Parámetros de Salida`
  - `a`: Forma factorizada en función de la tabla mostrada.
  - `scale`: Número en coma flotante que indica los valores propios que debieran ser escalados para compensar el escalado en esta rutina. Actualmente, siempre se devuelve `scale=1` pero se ha reservado en el interfaz, para una futura implementación.
  - `info`: Resultado global de la ejecución
    - `= 0`: Ejecución con éxito
    - `<0`: Si el argumento i-esimo es una matriz y la entrada j-esima tuvo un valor ilegal, entoces `info=-(i*100+j)`. Si el argumento i-esimo es un escalar y tuvo un valor ilegal entonces `info=-i`.
    - `<0`: Si `info=k`. La factorización ha sido completada pero el factor `U` es singular por lo que la solución no pudo ser calculada.

A continuación mostramos un ejemplo en la utilización de esta rutina:

```
from  PyACTS import *
import PyACTS.PyScaLAPACK as PySLK
from RandomArray import *
from Numeric import *
n,nrhs=8,2
#Initiliaze the Grid
PyACTS.gridinit(nb=2)
if PyACTS.iread==1:
print "Ejemplo de Utilizacion ScaLAPACK: PvSYGST"
print "N=",n,";nprow x npcol:",PyACTS.nprow,"x",PyACTS.npcol
print "Tam. Bloques:",PyACTS.mb,"*",PyACTS.nb
a=8*identity(n,Float)+ones([n,n],Float)
b=identity(n,Float)
print "a=",a
else:
a,b=None,None
#We convert Numeric Array to PyACTS.Scalapack Array
ACTS_lib=1 # 1=Scalapack
a=Num2PyACTS(a,ACTS_lib)
b=Num2PyACTS(b,ACTS_lib)
#We call ScaLAPACK routine
b,info= PySLK.pvpotrf(b,uplo="U")
a,scale,info= PySLK.pvsygst(a,b)
a=PyACTS2Num(a)
if PyACTS.iread==1:
print " Q' * sub( A ) * Q = T-->",a
print "scale:",scale
print "Info:",info
PyACTS.gridexit()
```