Num Trait

The num trait exists to restrict types to exclusively numeric types.

```
use std::ops::{Add, Sub, Mul, Div, Rem, Neg};

/// A custom `Num` trait to restrict types to basic numeric operations
pub trait Num:
    Add<Output = Self>
    + Sub<Output = Self>
    + Mul<Output = Self>
    + Div<Output = Self>
    + Rem<Output = Self>
    + PartialOrd
    + Copy
    + Default
    + 'static
{
    /// Method to return the additive identity (zero)
    fn zero() -> Self;

    /// Method to return the multiplicative identity (one)
    fn one() -> Self;
}

// Implement `Num` for primitive integer and floating-point types
macro_rules! impl_num_for_primitive {
    ($($t:ty)*) => ($(
        impl Num for $t {
            #[inline]
            fn zero() -> Self { 0 as $t }

            #[inline]
            fn one() -> Self { 1 as $t }
        }
    )*)
}

impl_num_for_primitive!(u8 u16 u32 u64 u128 i8 i16 i32 i64 i128 f32 f64);
```

Vector and Matrix type

```
pub struct NRvector<T: Num> {
    v: Vec<T>,
}

impl<T: Num> NRvector<T> {
    // Default constructor
    pub fn new() -> Self {
```

```
        NRvector { v: Vec::new() }
    }

    // Construct vector of size n
    pub fn with_size(n: usize) -> Self
    where
        T: Default + Clone,
    {
        NRvector {
            v: vec![T::default(); n],
        }
    }

    // Initialize to constant value a
    pub fn with_value(n: usize, a: T) -> Self
    where
        T: Clone,
    {
        NRvector { v: vec![a; n] }
    }

    // Initialize to values in a slice (C-style array equivalent)
    pub fn from_slice(a: &[T]) -> Self
    where
        T: Clone,
    {
        NRvector { v: a.to_vec() }
    }

    // Copy constructor
    pub fn copy_from(rhs: &NRvector<T>) -> Self
    where
        T: Clone,
    {
        NRvector { v: rhs.v.clone() }
    }

    // Assignment operator
    pub fn assign(&mut self, rhs: &NRvector<T>)
    where
        T: Clone,
    {
        self.v = rhs.v.clone();
    }

    // Return size of vector
    pub fn size(&self) -> usize {
        self.v.len()
    }

    // Resize, losing contents
    pub fn resize(&mut self, newn: usize)
    where
        T: Default + Clone,
```

```
    {
        self.v.resize(newn, T::default());
    }

    // Resize and assign `a` to every element
    pub fn assign_with_value(&mut self, newn: usize, a: T)
    where
        T: Clone,
    {
        self.v = vec![a; newn];
    }
}

// Indexing
impl<T: Num> Index<usize> for NRvector<T> {
    type Output = T;

    fn index(&self, index: usize) -> &Self::Output {
        &self.v[index]
    }
}

impl<T: Num> IndexMut<usize> for NRvector<T> {
    fn index_mut(&mut self, index: usize) -> &mut Self::Output {
        &mut self.v[index]
    }
}


pub struct NRmatrix<T: Num> {
    data: Vec<T>,
    shape: [usize; 2], // shape[0] = rows, shape[1] = columns
}

impl<T: Num> NRmatrix<T> {
    /// Default constructor: creates an empty matrix
    pub fn new() -> Self {
        NRmatrix {
            data: Vec::new(),
            shape: [0, 0],
        }
    }

    /// Construct an n x m matrix with default values
    pub fn with_size(rows: usize, cols: usize) -> Self
    where
        T: Default + Clone,
    {
        NRmatrix {
            data: vec![T::default(); rows * cols],
            shape: [rows, cols],
        }
    }
```

```rust
    /// Initialize to constant value `a`
    pub fn with_value(rows: usize, cols: usize, value: T) -> Self
    where
        T: Clone,
    {
        NRmatrix {
            data: vec![value; rows * cols],
            shape: [rows, cols],
        }
    }

    /// Initialize to values in a flattened slice (C-style array
equivalent)
    pub fn from_slice(rows: usize, cols: usize, slice: &[T]) -> Self
    where
        T: Clone,
    {
        assert_eq!(slice.len(), rows * cols, "Slice length does not match
matrix dimensions");
        NRmatrix {
            data: slice.to_vec(),
            shape: [rows, cols],
        }
    }

    /// Copy constructor
    pub fn copy_from(rhs: &NRmatrix<T>) -> Self
    where
        T: Clone,
    {
        NRmatrix {
            data: rhs.data.clone(),
            shape: rhs.shape,
        }
    }

    /// Assignment operator
    pub fn assign(&mut self, rhs: &NRmatrix<T>)
    where
        T: Clone,
    {
        self.data = rhs.data.clone();
        self.shape = rhs.shape;
    }

    /// Return the number of rows
    pub fn nrows(&self) -> usize {
        self.shape[0]
    }

    /// Return the number of columns
    pub fn ncols(&self) -> usize {
        self.shape[1]
    }
```

```rust
    /// Resize, losing contents
    pub fn resize(&mut self, new_rows: usize, new_cols: usize)
    where
        T: Default + Clone,
    {
        self.data.resize(new_rows * new_cols, T::default());
        self.shape = [new_rows, new_cols];
    }

    /// Resize and assign `value` to every element
    pub fn assign_with_value(&mut self, new_rows: usize, new_cols: usize,
value: T)
    where
        T: Clone,
    {
        self.data = vec![value; new_rows * new_cols];
        self.shape = [new_rows, new_cols];
    }

    /// Access an element by (row, column)
    pub fn get(&self, row: usize, col: usize) -> &T {
        assert!(row < self.shape[0] && col < self.shape[1], "Index out of
bounds");
        &self.data[row * self.shape[1] + col]
    }

    /// Mutable access to an element by (row, column)
    pub fn get_mut(&mut self, row: usize, col: usize) -> &mut T {
        assert!(row < self.shape[0] && col < self.shape[1], "Index out of
bounds");
        &mut self.data[row * self.shape[1] + col]
    }
}

// Indexing for row access
impl<T: Num> Index<usize> for NRmatrix<T> {
    type Output = [T];

    fn index(&self, row: usize) -> &Self::Output {
        let start = row * self.shape[1];
        let end = start + self.shape[1];
        &self.data[start..end]
    }
}

impl<T: Num> IndexMut<usize> for NRmatrix<T> {
    fn index_mut(&mut self, row: usize) -> &mut Self::Output {
        let start = row * self.shape[1];
        let end = start + self.shape[1];
        &mut self.data[start..end]
    }
}
```

Error

```rust
/// Simple error handling (similar to the default `#define throw` macro in
C++)
macro_rules! simple_throw {
    ($message:expr) => {{
        eprintln!(
            "ERROR: {}\n in file {} at line {}",
            $message,
            file!(),
            line!()
        );
        std::process::exit(1);
    }};
}

/// Advanced error class `NRerror` equivalent
/// ```rust
/// // Example function that uses `throw!` for error handling
/// fn cholesky_example(success: bool) -> Result<(), NRerror> {
///     if !success {
///         throw!("Cholesky error occurred");
///     }
///     println!("Cholesky computation succeeded.");
///     Ok(())
///}
///
/// ```
#[derive(Debug)]
pub struct NRerror {
    pub message: String,
    pub file: &'static str,
    pub line: u32,
}

impl NRerror {
    pub fn new(message: &str, file: &'static str, line: u32) -> Self {
        NRerror {
            message: message.to_string(),
            file,
            line,
        }
    }
}

impl std::fmt::Display for NRerror {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "ERROR: {}\n in file {} at line {}",
            self.message, self.file, self.line
```

```
        )
    }
}

/// Function to handle the NRerror (equivalent to `NRcatch` in C++)
pub fn nrcatch(err: NRerror) {
    eprintln!("{}", err);
    std::process::exit(1);
}

/// Advanced error handling macro
macro_rules! advanced_throw {
    ($message:expr) => {
        Err(NRerror::new($message, file!(), line!()))
    };
}

/// Unified `throw!` macro that chooses between simple and advanced error
handling
macro_rules! throw {
    ($message:expr) => {
        if USE_NR_ERROR_CLASS {
            return advanced_throw!($message);
        } else {
            simple_throw!($message);
        }
    };
}
```

Solutions of Linear Algebraic Equations

```
/// A matrix structure for numerical operations using a single vector for
data storage.
///
/// # Example
/// ```rust
/// use crate::nr3::NRmatrix;
///
///
/// let mut a = NRmatrix::with_value(3, 3, 1.0);
/// *a.get_mut(0, 0) = 2.0;
/// *a.get_mut(0, 1) = -1.0;
/// *a.get_mut(0, 2) = 0.0;
/// *a.get_mut(1, 0) = -1.0;
/// *a.get_mut(1, 1) = 2.0;
/// *a.get_mut(1, 2) = -1.0;
/// *a.get_mut(2, 0) = 0.0;
/// *a.get_mut(2, 1) = -1.0;
/// *a.get_mut(2, 2) = 2.0;
///
```

```
/// let mut b = NRmatrix::with_value(3, 1, 0.0);
/// *b.get_mut(0, 0) = 1.0;
/// *b.get_mut(1, 0) = 2.0;
/// *b.get_mut(2, 0) = 3.0;
///
/// gaussj(&mut a, &mut b);
///
/// println!("Inverted matrix a: {:?}", &a);
/// for i in 0..a.nrows() {
///     for j in 0..a.ncols() {
///         print!("{:.3} ", a.get(i, j));
///     }
///     println!();
/// }
///
/// println!("Solution matrix b:");
/// for i in 0..b.nrows() {
///     for j in 0..b.ncols() {
///         print!("{:.3} ", b.get(i, j));
///     }
///     println!();
/// }
/// ```
pub fn gaussj(a: &mut NRmatrix<f64>, b: &mut NRmatrix<f64>) {
    let n = a.nrows();
    let m = b.ncols();

    let mut indxc = vec![0; n];
    let mut indxr = vec![0; n];
    let mut ipiv = vec![0; n];

    for i in 0..n {
        let mut big = 0.0;
        let mut irow = 0;
        let mut icol = 0;

        for j in 0..n {
            if ipiv[j] != 1 {
                for k in 0..n {
                    if ipiv[k] == 0 {
                        let abs_a = a.get(j, k).abs();
                        if abs_a >= big {
                            big = abs_a;
                            irow = j;
                            icol = k;
                        }
                    }
                }
            }
        }

        ipiv[icol] += 1;

        if irow != icol {
```

```
            a.swap_rows(irow, icol);
            b.swap_rows(irow, icol);
        }

        indxr[i] = irow;
        indxc[i] = icol;

        if *a.get(icol, icol) == 0.0 {
            panic!("gaussj: Singular Matrix");
        }

        let pivinv = 1.0 / *a.get(icol, icol);
        *a.get_mut(icol, icol) = 1.0;

        for l in 0..n {
            *a.get_mut(icol, l) *= pivinv;
        }

        for l in 0..m {
            *b.get_mut(icol, l) *= pivinv;
        }

        for ll in 0..n {
            if ll != icol {
                let dum = *a.get(ll, icol);
                *a.get_mut(ll, icol) = 0.0;

                for l in 0..n {
                    *a.get_mut(ll, l) -= *a.get(icol, l) * dum;
                }

                for l in 0..m {
                    *b.get_mut(ll, l) -= *b.get(icol, l) * dum;
                }
            }
        }
    }

    for l in (0..n).rev() {
        if indxr[l] != indxc[l] {
            a.swap_cols(indxr[l], indxc[l]);
        }
    }
}
```

LU Decomposition

```
use crate::nr3::NRmatrix;

/// LU Decomposition object for solving linear equations and related
operations.
```

```rust
/// ```rust
/// let mut a = NRmatrix::with_value(3, 3, 1.0);
/// *a.get_mut(0, 0) = 2.0;
/// *a.get_mut(0, 1) = -1.0;
/// *a.get_mut(0, 2) = 0.0;
/// *a.get_mut(1, 0) = -1.0;
/// *a.get_mut(1, 1) = 2.0;
/// *a.get_mut(1, 2) = -1.0;
/// *a.get_mut(2, 0) = 0.0;
/// *a.get_mut(2, 1) = -1.0;
/// *a.get_mut(2, 2) = 2.0;
///
/// let b = vec![1.0, 2.0, 3.0];
/// let mut x = vec![0.0; 3];
///
/// let lu = LUdcmp::new(&a);
/// lu.solve(&b, &mut x);

/// println!("Solution vector x:");
/// for xi in &x {
///     println!("{:.3}", xi);
/// }```
pub struct LUdcmp {
    n: usize,
    lu: NRmatrix<f64>,
    indx: Vec<usize>,
    d: f64,
}

impl LUdcmp {
    /// Constructor. Takes a matrix `a` and computes its LU decomposition.
    pub fn new(a: &NRmatrix<f64>) -> Self {
        let n = a.nrows();
        let mut lu = a.clone();
        let mut indx = vec![0; n];
        let mut d = 1.0;
        let mut vv = vec![0.0; n];
        const TINY: f64 = 1.0e-40;

        for i in 0..n {
            let mut big = 0.0;
            for j in 0..n {
                let temp = lu.get(i, j).abs();
                if temp > big {
                    big = temp;
                }
            }
            if big == 0.0 {
                panic!("Singular matrix in LUdcmp");
            }
            vv[i] = 1.0 / big;
        }

        for k in 0..n {
```

```
            let mut big = 0.0;
            let mut imax = k;

            for i in k..n {
                let temp = vv[i] * lu.get(i, k).abs();
                if temp > big {
                    big = temp;
                    imax = i;
                }
            }

            if k != imax {
                lu.swap_rows(imax, k);
                d = -d;
                vv.swap(imax, k);
            }

            indx[k] = imax;
            if *lu.get(k, k) == 0.0 {
                *lu.get_mut(k, k) = TINY;
            }

            for i in k + 1..n {
                let temp = *lu.get_mut(i, k) / *lu.get(k, k);
                for j in k + 1..n {
                    *lu.get_mut(i, j) -= temp * *lu.get(k, j);
                }
            }
        }

        LUdcmp { n, lu: lu.clone(), indx, d }
    }

    /// Solves for a single right-hand side.
    pub fn solve(&self, b: &[f64], x: &mut [f64]) {
        if b.len() != self.n || x.len() != self.n {
            panic!("LUdcmp::solve bad sizes");
        }

        for i in 0..self.n {
            x[i] = b[i];
        }

        let mut ii = 0;
        for i in 0..self.n {
            let ip = self.indx[i];
            let mut sum = x[ip];
            x[ip] = x[i];
            if ii != 0 {
                for j in 0..i {
                    sum -= self.lu.get(i, j) * x[j];
                }
            } else if sum != 0.0 {
                ii = i + 1;
```

```
            }
            x[i] = sum;
        }

        for i in (0..self.n).rev() {
            let mut sum = x[i];
            for j in i + 1..self.n {
                sum -= self.lu.get(i, j) * x[j];
            }
            x[i] = sum / self.lu.get(i, i);
        }
    }

    /// Solves for multiple right-hand sides.
    pub fn solve_matrix(&self, b: &NRmatrix<f64>, x: &mut NRmatrix<f64>) {
        let m = b.ncols();
        if b.nrows() != self.n || x.nrows() != self.n || b.ncols() !=
x.ncols() {
            panic!("LUdcmp::solve_matrix bad sizes");
        }

        let mut xx = vec![0.0; self.n];
        for j in 0..m {
            for i in 0..self.n {
                xx[i] = *b.get(i, j);
            }
            self.solve(&xx, &mut xx.clone());
            for i in 0..self.n {
                *x.get_mut(i, j) = xx[i];
            }
        }
    }
}
```